FreeCampus

THE COMPLETE STUDY BOOK

Python Foundations

From your first program
to practical, tested Python projects.

4 milestones / 16 units / 97 lessons / 20 challenges

Printable checkpoints, answers, and offline study guidance

Curriculum 22 • Free to learn. Open to improve.

freecampus.org/python

About this edition

This is the complete Python Foundations course from FreeCampus: one connected sequence of 4 milestones, 16 units, 97 lessons, 16 unit challenges, and 4 milestone challenges. It starts with no programming experience and continues through practical Python projects, testing, packaging, command-line applications, and documentation.

This PDF is generated from the same lesson files as the website. Curriculum version **22** is printed here; the website may receive later corrections. Use the online edition when you need the latest material.

The book is designed for focused reading, annotation, and printing. It includes all lesson explanations and code, printable checkpoints, visible hints and solutions, and a separate checkpoint answer key. Diagrams are adapted for the page; labeled transitions are also written below them.

Code is printed as source, not executed by the PDF build. Expected output and intentional error examples remain part of the lessons. A small continuation arrow at the end of a long code line means the line wraps on paper; it is not Python syntax. Do not type the arrow when copying code.

Every lesson begins with a link to its online page, where you can open its matching Colab notebook. A PDF cannot run code, grade a quiz, or update browser progress. The study record at the end is for your own notes and is not a grade, verified completion record, or certificate. The final project and certificate are planned, not included as available credentials.

FreeCampus makes rigorous learning materials open to everyone. You can study at your own pace, share the course, and contribute corrections. The complete BSD 3-Clause license and source link appear at the end of the book.

How to study with this book

Read with a working notebook beside you

You only need a browser to begin. Open the lesson link and choose its Colab notebook, or type the examples into your own Python environment. Later units explain dependable local workflows and complete projects.

The course estimates **388–556 hours** of active study. That includes typing, predicting, explaining, exercises, quizzes, and debugging. It is not a reading-time estimate. A long lesson can take several sessions; stop at a checkpoint and note where to resume.

Use a repeatable practice cycle

1. Read the task and predict the result before running its code.
2. Run the example and compare the exact output with your prediction.
3. Explain what each relevant line reads, changes, returns, or displays.
4. Make one controlled change and predict again.
5. Answer the checkpoint on paper before opening the answer key.
6. Complete the lab and then the unit challenge with your own evidence.
7. At each milestone boundary, complete its challenge to connect the units.

The PDF prints hints and solutions because paper cannot hide expandable panels. Try the task first, and cover the solution while you work if that helps. A correct answer is useful only when you can explain why it is correct.

Keep code, output, and notes separate

Copy code into a code cell or source file; keep observed output and explanations in your notes. Preserve indentation, quotation marks, and Unicode characters. When a printed line wraps, join it into one source line unless the lesson shows an explicit multi-line construct.

Examples marked as intentional failures are for repair practice. Read the lesson’s explanation before treating an error example as working code.

Keep a short debugging record

Write down what you expected, what happened, your hypothesis, the one change you made, and the result of running again. At the end of a unit, use the study record to note what you can explain and what needs another pass.

Online progress remains local to the browser in which you record it. Reading this book does not automatically mark lessons complete on the website.

Contents

About this edition 2

How to study with this book 3

Contents 4

Write and Explain Your First Programs 8

Unit 0: Get Started . . . 9

Meet Python Through a Tiny Program . . . 13

Learn with Predictions and Evidence . . . 23

Use AI While Keeping the Thinking Yours . . . 34

Work Reliably in Colab and Jupyter . . . 44

Build a Dependable Local Python Workspace . . . 55

Unit Challenge: Launch Your Python Learning Lab . . . 68

Unit 1: Learning to Read and Write Python Code . . . 72

Values, Names, and Assignment . . . 76

Expressions, Statements, and Execution Order . . . 91

Python's Words and Symbols . . . 106

Indentation and Code Blocks . . . 121

Writing Python Across Multiple Lines . . . 142

Comments and Docstrings That Help . . . 158

Reading and Fixing Syntax Errors . . . 172

Unit Challenge: Repair and Explain a Broken Study Planner . . . 188

Unit 2: Numbers, Text, and Other Values . . . 198

Values, Types, Conversion, and None . . . 203

Integers and Everyday Arithmetic . . . 215

Floating-Point Numbers and Precision . . . 225

Money, Exact Ratios, and Complex Numbers . . . 236

Strings and Unicode Text . . . 247

Input, Output, and Clear Formatting . . . 261

Booleans, Comparisons, and Truth . . . 273

Dates, Times, and Durations . . . 285

Unit Challenge: Build a Date-Aware Purchase Summary . . . 297

Milestone Challenge: Pack a Moon-Garden Supply Box . . . 306

Turn Data into Reusable Decisions 308

Unit 3: Organizing and Traversing Collections . . . 309

Lists: Keeping Values in Order . . . 315

Tuples and Unpacking Fixed Records . . . 327
Dictionaries: Finding Values by Key . . . 337
Sets: Uniqueness and Group Comparisons . . . 349
Visiting Items in a Collection . . . 360
Sorting Collections Without Losing Meaning . . . 372
Building and Reading Nested Data . . . 382
Unit Challenge: Decode the Starship Signal Vault . . . 396
Unit 4: Making Decisions and Repeating Work . . . 407
Choosing a Path with `if`, `elif`, and `else` . . . 412
Combining Conditions Without Surprises . . . 423
Turning Rules into Complete Decisions . . . 434
Building Results with `for` Loops . . . 444
Repeating Until Something Changes . . . 457
Searching, Skipping, and Stopping Loops . . . 468
Working Through Grids and Pairs . . . 478
Building Collections with Comprehensions . . . 488
Unit Challenge: Escape the Clockwork Maze . . . 499
Unit 5: Building Reusable Functions . . . 509
Defining Functions and Returning Useful Results . . . 516
Designing Clear Parameters and Calls . . . 527
Breaking a Program into Small Functions . . . 539
Following Names Through Function Calls . . . 550
Passing Functions as Values . . . 561
Solving Smaller Problems with Recursion . . . 572
Producing Values One at a Time . . . 583
Adding Behavior with Decorators . . . 593
Unit Challenge: Unlock the Moonlit Library . . . 605
Unit 6: Sharing, Changing, and Copying Objects . . . 615
When Names Share the Same Object . . . 620
Changing Objects and Choosing Safe Keys . . . 630
Copying Nested Data Without Surprises . . . 641
Designing Functions That Control Mutation . . . 652
Unit Challenge: Break the Mirror Vault . . . 664
Milestone Challenge: Route Supplies through a Puzzle Village . . . 674

Build Reliable Programs from External Data 676

Unit 7: Solving Problems with Algorithms . . . 677
Turn a Vague Idea into Testable Examples . . . 683
Turn a Plan into Working Python . . . 695
Choose an Algorithm That Fits . . . 707
Know It Works and Understand the Cost . . . 722
Unit Challenge: Wake the Clockwork Observatory . . . 737

Unit 8: Errors, Exceptions, and Debugging . . . 747
Understand Failures and Read Tracebacks . . . 752
Raise Useful Errors and Handle Expected Failures . . . 765
Debug One Hypothesis at a Time . . . 777
Pause, Inspect, and Shrink a Bug . . . 789
Unit Challenge: Repair the Glitched Arcade . . . 800
Unit 9: Files, Paths, and External Data . . . 810
Find, Read, and Write Files Reliably . . . 816
Keep Text Intact Across Encodings and Newlines . . . 828
Move Records Between CSV, JSON, YAML, and Python . . . 839
Turn Messy Text into Trustworthy Records . . . 853
Cross the Text–Binary Boundary Safely . . . 865
Unit Challenge: Restore the Starship Signal Archive . . . 876
Unit 10: Modules, Environments, and Python Projects . . . 891
Find the Right Tool Without Memorizing Everything . . . 896
Turn One Script into Reusable Modules . . . 908
Design a Package People Can Rely On . . . 921
Build a Project You Can Navigate and Share . . . 934
Give Each Project a Clean Python Environment . . . 947
Build and Verify an Installable Python Project . . . 960
Unit Challenge: Release the Moonlight Cipher Kit . . . 975
Milestone Challenge: Recover the Observatory Log . . . 989

Ship a Tested and Documented Python Tool 991

Unit 11: Object-Oriented Python and Dataclasses . . . 992
Build Your First Useful Class . . . 998
Keep Objects in Valid States . . . 1012
Use Dataclasses for Value-Like Objects . . . 1025
Choose Composition Before Inheritance . . . 1038
Model a Complete Road-Crossing Game . . . 1053
Unit Challenge: Run the Tiny Robot Tournament . . . 1070
Unit 12: Command-Line Applications . . . 1085
Design a Friendly Command with argparse . . . 1090
Put Output, Errors, and Exit Status in the Right Place . . . 1104
Resolve Configuration Without Exposing Secrets . . . 1117
Write Useful Logs Without Polluting Output . . . 1132
Keep the Command Thin and the Core Reusable . . . 1143
Unit Challenge: Decode the Deep-Sea Beacon . . . 1158
Unit 13: Testing Python Programs . . . 1172
Write Tests That Protect Behavior . . . 1178
Run Pytest and Read the Failure . . . 1191
Turn Examples into Clear Parametrized Cases . . . 1205

Build an Isolated Test World with Fixtures . . . 1217
Control Dependencies Without Fragile Mocks . . . 1231
Test at the Smallest Boundary That Can Fail . . . 1245
Preserve Bugs and Explore General Rules . . . 1257
Unit Challenge: Clear the Buggy Spaceport for Launch . . . 1270
Unit 14: Code Quality and Maintainability . . . 1286
Write Python People Can Follow . . . 1291
Make Contracts Visible with Type Annotations . . . 1303
Read MyPy Errors and Close Type Gaps . . . 1314
Format and Lint a Project with Ruff . . . 1325
Catch Small Problems Before Commit . . . 1335
Repeat the Quality Gate in CI . . . 1345
Refactor in Small, Verified Steps . . . 1356
Unit Challenge: Untangle the Clockwork Observatory . . . 1368
Unit 15: Documentation as Part of the Product . . . 1380
Design Documentation Around Reader Needs . . . 1384
Write Instructions People Can Follow . . . 1394
Document a Public Python API . . . 1404
Keep Small Examples Honest with Doctest . . . 1415
Review Documentation Like Code . . . 1426
Build a Documentation Site with Quarto . . . 1437
Publish Documentation and Keep It Trustworthy . . . 1448
Unit Challenge: Publish the Puzzle Garden Field Guide . . . 1459
Milestone Challenge: Publish a Tiny Community Library Tool . . . 1471

Checkpoint answer key **1473**

Study record **1531**

License and source **1532**

Font licenses **1533**

Write and Explain Your First Programs

Online lesson and runnable notebook

Write and explain a dispatch-card program using names, integer arithmetic, conversion, and formatting.

What you will build

Write and explain a dispatch-card program using names, integer arithmetic, conversion, and formatting.

Follow the units in order and complete each unit challenge. Keep examples and debugging notes that help explain your choices. This overview is orientation, not a scored activity.

Unit sequence

1. Get Started — 5 lessons and a unit challenge.
2. Learning to Read and Write Python Code — 7 lessons and a unit challenge.
3. Numbers, Text, and Other Values — 8 lessons and a unit challenge.

Prepare and pace your work

Review the prerequisites in the course overview before beginning this first milestone. Use each unit's active-time estimate and pause at lesson boundaries. Plan another 2–4 hours for the milestone challenge, including rerunning checks and explaining your result.

Use Colab for notebook exercises and the workspace introduced in the units for local environment, packaging, and command-line tasks.

Connect the units

Finish with Pack a Moon-Garden Supply Box. Combine the milestone skills and check a changed condition. The challenge provides a contract, expected results, a hint, and self-review guidance.

Community learning is ungraded. Completion controls record your own progress only in this browser on this device; they do not issue a verified credential.

Begin the first unit.

Unit 0: Get Started

Online lesson and runnable notebook

Prepare a dependable Python learning lab, practise from observable evidence, and choose browser or local tools without assuming previous programming experience.

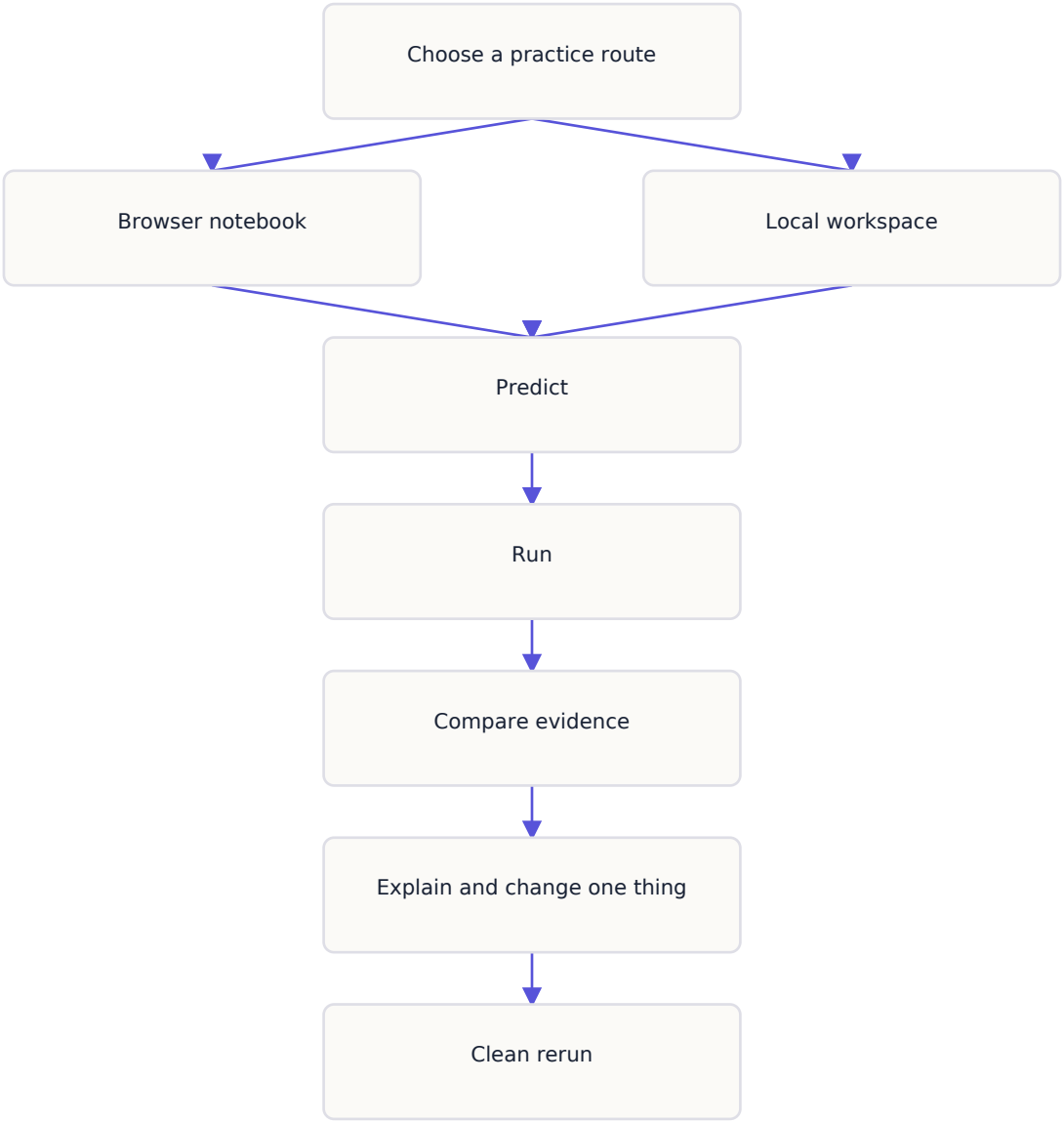
- **Level:** Python Foundations · Unit 0
- **Estimated time:** 13–20 hours including the challenge
- **Unit outcome:** Run and explain a small Python program, keep trustworthy evidence in a clean notebook, choose a suitable practice environment, diagnose common first-run problems, and use AI assistance without surrendering prediction or verification.
- **Practice in:** Google Colab or Jupyter, then a local editor and terminal when your device permits

1. Start with a working place to learn

You do not need programming experience before this unit. You do not need to memorize Python words, choose a career specialty, or install every tool you have heard about. You need one place where you can run a small program, observe what happens, preserve useful notes, and try again without hiding the evidence.

The fastest route is a browser notebook such as Google Colab. It lets you begin without installing Python. A local workspace becomes important when you need to work with saved files, project folders, terminals, editors, and isolated environments. This unit teaches both routes. If a school or work computer blocks installation, you will record that constraint, continue in Colab, and identify what access you will need before later project units.

Notice that both routes lead to the same learning loop and a clean rerun.



You will run Python in the first lesson. Formal details about names, values, expressions, functions, and errors come later. When one of those shapes appears here, the lesson explains only what you need for the current task and points forward to the unit that develops it.

2. Follow the five-step sequence

Step	Lesson	What you will practise or create
1	Meet Python Through a Tiny Program	Run `first_run.py`, connect each source line to its output, change one message, and repair a missing delimiter.
2	Learn with Predictions and Evidence	Build `learning-log.md` and use observations rather than guesses to decide the next change.
3	Use AI While Keeping the Thinking Yours	Review a supplied AI answer, verify it locally, protect private information, and retain your own conclusion.
4	Work Reliably in Colab and Jupyter	Create `first_notebook.ipynb`, expose hidden state, and prove a clean top-to-bottom run.
5	Build a Dependable Local Python Workspace	Run the saved script from a known folder, align the editor and interpreter, verify `.venv`, and finish `launch-card.md`.
6	Unit Challenge	Complete three scenario rounds and confirm that your learning lab is ready for Unit 1.

Plan 13–20 hours of active work for this unit. Each lesson's final lab is a natural stopping point. Save your work and its observed result before leaving, then rerun the nearest check when you return.

Each lesson asks you to predict, run, compare, explain, modify, and repair, but those words will not appear as one generic practice section copied five times. You will use them on different evidence: source code, an experiment record, an AI claim, notebook history, and local paths.

3. Prepare without buying anything

Bring:

- a modern web browser;
- a place to save a notebook or download files;
- permission to create a small folder if you will practise locally; and
- a plain-text note for commands, outputs, and questions.

The course does not require a paid editor, notebook service, or AI subscription. You can complete the AI lesson using the supplied responses. Do not paste passwords, API keys, private repository code, personal records, or assessment answers into an external service.

Choose one route before adding tools

Begin in Colab if installation would delay your first run. Begin locally if Python is already available and you can create files. Do not install Python, Conda, Anaconda, multiple editors, and several notebook systems at once. One working route creates better evidence than five half-configured routes. :::

4. Know what success looks like

You are ready for Unit 1 when you can:

1. point to the source line responsible for each line of a tiny program’s output;
2. keep prediction, observation, explanation, and next change separate;
3. treat an AI response as a claim to verify rather than permission to stop thinking;
4. restart a notebook and run every cell from the top without relying on hidden state;
5. state which Python command and environment you are using; and
6. preserve the exact command, output, and relevant file when you need help.

Progress in this site is local and self-reported. It is not a verified exam, identity check, or certificate record. A Foundations final project and certificate are planned but are not available yet; they will be announced later.

Begin with Meet Python Through a Tiny Program

Meet Python Through a Tiny Program

Online lesson and runnable notebook

Run a complete three-line Python program, connect source lines to exact output, make a controlled change, and repair a first syntax error.

- **Level:** Beginner · no programming experience assumed
- **Estimated time:** 2–3 hours
- **You will learn:** Distinguish source code, the Python interpreter, output, and an error by running and repairing a tiny complete program.
- **Practice in:** Google Colab, JupyterLab, or a local editor

Your Python Learning Lab begins with something observable: three lines of source code that should produce three exact lines of output. You will not study every symbol formally yet. You will learn enough to run the program, follow its order, and make one defensible change.

Questions to answer

- What is the difference between a program, source code, Python, and the Python interpreter?
- Which source line produces each output line?
- What does Python do when it reaches an instruction it cannot read?
- How can one small edit create a predictable change?
- What evidence should you preserve when the first run fails? :::

1. Give Python one complete instruction

A **program** is a set of instructions for a computer. **Source code** is the text in which those instructions are written. Python is the language used here; the **Python interpreter** is software that attempts to run Python source code.

Start with one instruction:

```python
print("Python Learning Lab")
```

Predict before running: what text will appear, and will the quotation marks be part of the displayed result?

Expected output:

```text
Python Learning Lab
```

The source contains more than the output:

- `print` asks Python to display a value;

- `(` and `)` enclose what will be supplied to `print`;
- the quotation marks mark the beginning and end of text in the source; and
- `Python Learning Lab` is the text value displayed.

The quotation marks help Python read the source. They are not part of this output. Unit 1 studies the words and symbols in much more detail, and Unit 2 studies text values. For now, recognize the complete visible shape `print("message")`.

Try three valid messages separately:

```python
print("Ready")
```

```python
print("Ready for Python")
```

```python
print("Ready: tools checked")
```

Spaces and punctuation *inside* the quotes belong to the message. Compare the exact outputs rather than saying only that each example “worked.”

2. Follow instructions from top to bottom

Now run the first Learning Lab program:

```python
print("Python Learning Lab")
print("Status: ready")
print("Next step: predict before running")
```

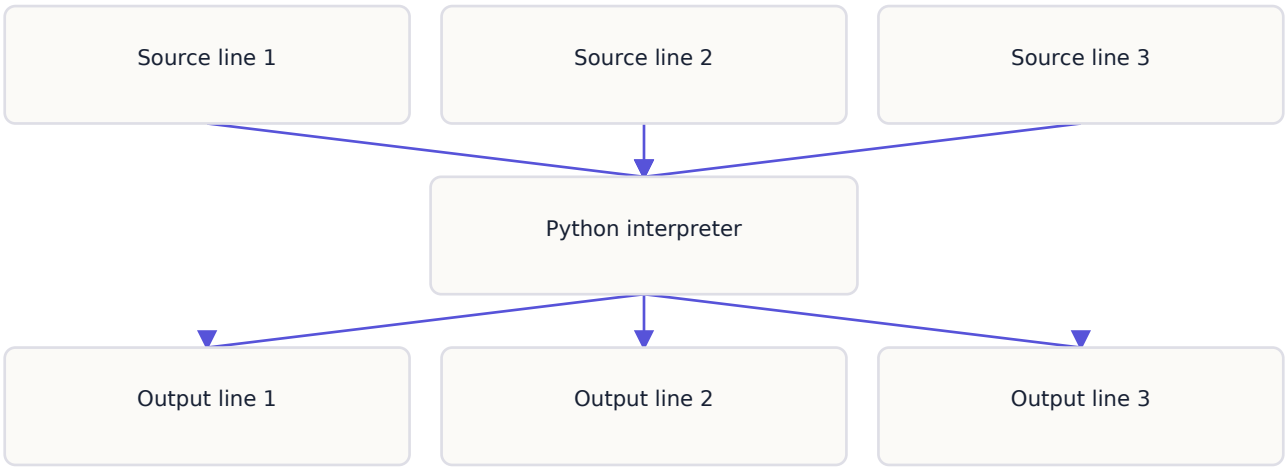
Write all three predicted output lines before executing it. Python normally attempts this short script from the first visible instruction to the last:

Source position	Instruction’s message	Expected output position
1	`Python Learning Lab`	1
2	`Status: ready`	2
3	`Next step: predict before running`	3

Expected output:

```text
Python Learning Lab
Status: ready
Next step: predict before running
```

Trace each source instruction through the interpreter to its output line.



Reorder the source deliberately:

```python
print("Next step: predict before running")
print("Python Learning Lab")
print("Status: ready")
```

The messages did not change, but the order did. Predict the complete output. Then run and check line 1, line 2, and line 3 separately. “The same text appeared” is incomplete evidence because order is part of the behavior.

Checkpoint: connect source to output

Source and output checkpoint

Choose the best answer for each question.

1. Which description best fits source code?

- **A.** The text containing instructions that a programmer writes
- **B.** Only the text displayed after a program runs
- **C.** A screenshot of an editor
- **D.** An automatic explanation created by Python

2. What does the Python interpreter do in this lesson?

- **A.** It guesses the learner’s unstated goal
- **B.** It attempts to run the Python source and reports output or an error
- **C.** It changes every message into a number
- **D.** It saves every notebook automatically

3. What is the second output line from print("Lab"); print("Ready"); print("Go") when the calls are on three separate source lines?

- **A.** Lab
- **B.** Go

- **C.** Ready
- **D.** print

4. For print("Status: ready"), what normally appears as output?

- **A.** print
- **B.** The quotation marks only
- **C.** Nothing
- **D.** Status: ready without the source quotation marks

Write your choices before checking the answer key at the end of this book.

3. Read code and output as different records

A common beginner mistake is to copy the output back into a code cell:

```
PYTHON
Python Learning Lab
Status: ready
```

Those lines are useful output, but they are not the original Python instructions. Keep each kind of record in its proper place:

```
PYTHON
print("Python Learning Lab")
print("Status: ready")
```

```
TEXT
Python Learning Lab
Status: ready
```

In a notebook, code belongs in a code cell and an explanation or copied result belongs in a Markdown cell. In a `.py` file, keep the Python source in the file and observe output in the terminal. Lesson 4 develops the notebook distinction.

Another misleading record shows source and guessed output without a label:

```
TEXT
print("Status: ready")
Status: ready
```

A future reader cannot tell whether the second line was observed or merely predicted. A better learning note is explicit:

```
TEXT
Source: print("Status: ready")
Prediction: Status: ready
Observed output: Status: ready
```

When prediction and observation match, preserve both. Matching evidence is still evidence. When they differ, do not edit the old prediction to pretend it matched; the difference tells you what to investigate.

4. Make one controlled change

Return to the understood program:

```python
print("Python Learning Lab")
print("Status: ready")
print("Next step: predict before running")
```

Change only `ready` to `checked`:

```python
print("Python Learning Lab")
print("Status: checked")
print("Next step: predict before running")
```

Before running, identify every expected downstream effect:

- output line 1 should remain unchanged;
- output line 2 should change from `Status: ready` to `Status: checked`; and
- output line 3 should remain unchanged.

This is a **controlled change**: one source difference gives you a clear claim to check. Contrast it with this edit:

```python
print("Learning Station")
print("Status: checked")
print("Next action: open Unit 1")
```

This program is valid, but all three messages changed. If the output surprises you, there are three possible source changes to inspect. Large changes are sometimes necessary; they are simply weaker experiments when you are learning one relationship.

Checkpoint: predict a controlled edit

Controlled change checkpoint

Choose the best answer for each question.

1. The only edit is changing print("Status: ready") to print("Status: checked"). Which output should change?

- **A.** Only the status output line
- **B.** Every output line
- **C.** No output line
- **D.** Only the program title

2. Three print calls keep the same messages but exchange positions. What behavior changes?

- **A.** The messages become invalid
- **B.** The order of the output lines

- **C.** Python stops displaying text
- **D.** The quotation marks appear in output

3. Which record best demonstrates a controlled experiment?

- **A.** Only the final screenshot
- **B.** Three edited messages with no original output
- **C.** Original source and output, one marked edit, predicted effect, and new output
- **D.** A copied program that was never run

4. The expected output has three lines in a stated order. Which check is strongest?

- **A.** Confirm that some text appeared
- **B.** Count only the source lines
- **C.** Check whether the editor stayed open
- **D.** Compare all three observed lines and their order with the expectation

Write your choices before checking the answer key at the end of this book.

5. Let the first error narrow the repair

Python must be able to recognize the source before it can follow the instructions. This example begins a text value with `"` but never closes it:

```
PYTHON
print("Status: ready)
```

Run it only after predicting that it will fail. The exact wording and caret may vary by Python version and environment, but the evidence will identify a `SyntaxError` and point near the unfinished text. Python is not saying your idea is bad. It is saying the source does not form a complete readable instruction.

Use a four-part first-error record:

Record	Example
Source attempted	`print("Status: ready)`
Observed evidence	`SyntaxError` near the unfinished text
Smallest hypothesis	The opening quote has no matching closing quote
One repair	Add `"` after `ready` and rerun

The repair is:

```
PYTHON
print("Status: ready")
```

Do not make unrelated edits before checking it. If you also rename the message, move the line, and replace punctuation, you will not know which edit repaired the readable shape.

A second case has balanced quotes but no closing parenthesis:

```python
print("Next step: predict before running"
```

Pair the visible delimiters:

1. opening parenthesis `(` needs closing parenthesis `)`;
2. opening quote `"` already has closing quote `"`;
3. therefore add only the missing `)`.

```python
print("Next step: predict before running")
```

A third case is valid Python but has an unintended message:

```python
print("Status: raedy")
```

It runs and displays `Status: raedy`. No syntax error can detect that spelling contradicts your intention. Compare output with the stated expectation, locate the first difference, change only the text, and rerun:

```python
print("Status: ready")
```

This distinction matters throughout programming: a run can succeed while the result is wrong for the task.

6. Preserve the evidence someone else needs

“Python does not work” is hard to investigate. A useful help request preserves:

- the environment, such as Colab or a local terminal;
- the complete smallest source that demonstrates the problem;
- the exact command or action used to run it;
- the complete output or error text;
- the result you expected; and
- the one repair already attempted.

For this source:

```python
print("Python Learning Lab")
print("Status: ready)
print("Next step: predict before running")
```

an effective note is:

```text
Environment: Google Colab code cell
Expected: three output lines
Observed: SyntaxError while reading line 2
```

TEXT (CONTINUED)

```
Smallest clue: line 2 opens text but does not close it
Next check: add the closing quote on line 2, then rerun unchanged lines 1 and 3
```

You do not yet need to understand a full traceback. Unit 8 teaches error and traceback diagnosis systematically. Here, learn to preserve exact evidence and repair the earliest visible incomplete instruction.

Checkpoint: repair the earliest failure

First repair checkpoint

Choose the best answer for each question.

1. Python reports a SyntaxError for print(“Ready). What is the smallest relevant repair?

- **A.** Add the closing quotation mark before the parenthesis
- **B.** Add two unrelated print calls
- **C.** Rename the file three times
- **D.** Delete the complete message

2. print("Status: raedy") runs, but the required output is Status: ready. What kind of evidence is this?

- **A.** Python could not read the source
- **B.** The source ran, but the observed result differs from the requirement
- **C.** The interpreter is definitely missing
- **D.** The notebook has no code cell

3. Which information makes a first-run help request most useful?

- **A.** Only the sentence It is broken
- **B.** Only a picture of the keyboard
- **C.** Smallest source, exact run action, exact evidence, expected result, and attempted repair
- **D.** A different large program

4. A three-line program stops at an incomplete line 2. What should you check before rewriting line 3?

- **A.** Whether every message uses different words
- **B.** Whether the browser theme is dark
- **C.** Whether a longer program exists online
- **D.** Whether repairing line 2 lets Python continue to the later line

Write your choices before checking the answer key at the end of this book.

7. Lab: create `first_run.py`

Create a folder named `python-learning-lab`. In Colab, you may place the source in one code cell now and download a `.py` copy after it works. Locally, create `first_run.py` inside the folder.

Your contract is exact:

```text
Python Learning Lab
Status: ready
Next step: predict before running
```

Complete these stages:

1. Write the three `print(...)` instructions.
2. Before running, copy the expected output into your learning notes.
3. Run the program and compare all three lines and their order.
4. Change only `ready` to `checked`; predict that only line 2 changes.
5. Observe the changed run, then restore `ready`.
6. Remove one closing quote deliberately, run once, and preserve the error.
7. Restore the quote and prove the original three-line output again.

Observable completion checks

Your final source has three complete calls. A clean run displays the three contract lines in order. Your notes contain the original prediction, the controlled-change prediction and result, and one error-and-repair record. :::

Hint: pair the visible shapes

Each line has `print(`, an opening quote, message text, a closing quote, and `)`. Compare one incomplete line with a complete line character by character.

Show one possible solution path

Create this final file:

```python
print("Python Learning Lab")
print("Status: ready")
print("Next step: predict before running")
```

Run it once unchanged. For the controlled experiment, make a copy or preserve the original result, replace only `ready` with `checked`, and rerun. Restore `ready`. For the repair record, remove only the quote after `ready`, capture the resulting `SyntaxError`, replace the quote, and rerun from the complete source.

Key points

- Source code is the program text; output is evidence produced by running it.
- The Python interpreter attempts complete instructions in visible order.
- Quotation marks and parentheses belong to the readable source shape even when they do not appear in output.
- One controlled edit supports a precise prediction about what should change and what should remain stable.
- A successful run can still produce the wrong result for the requirement.

- Preserve the smallest source, exact action, observed evidence, expectation, and one attempted repair when you need help.

References

- The Python tutorial: an informal introduction
- Python documentation glossary: interpreter
- Google Colab introduction

Learn with Predictions and Evidence

Online lesson and runnable notebook

Turn each small Python experiment into a trustworthy record of prediction, observation, explanation, controlled change, and repair.

- **Level:** Beginner
- **Estimated time:** 2.5–3.5 hours
- **You will learn:** Keep prediction, observation, explanation, experiment, and repair separate enough to learn from both expected and surprising runs.
- **Practice in:** Google Colab, JupyterLab, or a local editor plus `learning-log.md`

A program that runs once gives you a result. A result becomes learning evidence when you state what you expected, preserve what actually happened, explain the connection to the source, and use one controlled change to challenge that explanation. This lesson turns that cycle into a record you can revisit.

Questions to answer

- What makes a prediction precise enough to test?
- How are an observation and an explanation different?
- Why should one experiment change only one relevant cause?
- What should you record before asking for help?
- How can a surprising run improve your understanding instead of erasing it? :::

1. Make a prediction that could be wrong

Return to a complete program:

```python
print("Python Learning Lab")
print("Status: ready")
print("Next step: predict before running")
```

“Something will appear” is hard to disprove. A useful prediction states the observable behavior:

```text
Prediction
1. Python Learning Lab
2. Status: ready
3. Next step: predict before running
The lines will appear in this order with no quotation marks.
```

This prediction could be wrong in several visible ways: missing text, different punctuation, extra quotation marks, or a different order. That precision is a strength, not a risk. Learning requires a claim that the run can confirm or challenge.

Compare three predictions:

Prediction	Testable?	Why
“It should work.”	Weak	`work` does not identify an observable result.
“It will show ready.”	Partial	It names one fragment but not the complete behavior.
“Line 2 will be `Status: ready`; lines 1 and 3 will remain as stated.”	Strong	The output and stable boundaries are explicit.

Do not run first and write the result in the prediction box afterward. Put the prediction somewhere visible before execution: a Markdown cell, paper, or `learning-log.md`.

Predict punctuation, spacing, and order

Run this only after writing the exact one-line output:

```python
print("Status : ready")
```

`Status : ready` has a space before the colon. If your expected requirement is `Status: ready`, the code is readable but the result differs. Exact predictions train you to notice details that “looks about right” hides.

Now predict these two outputs separately:

```python
print("A")
print("B")
```

```python
print("B")
print("A")
```

Same instructions, different order, different behavior.

2. Keep observation separate from explanation

An **observation** reports what the run presented. An **explanation** connects that evidence to the source or environment.

```text
Observation: The output had Status: ready on line 2.
Explanation: The second source instruction is print("Status: ready"), and the
interpreter reached it after line 1.
```

The observation can be copied from the run. The explanation is a claim that could still be incomplete. Keeping them separate lets you improve the explanation without rewriting the evidence.

Here is a bad record:

```text
It printed ready because Python knew the lab was ready.
```

The run did display `ready`, but Python did not inspect your tools and make that decision. The literal message was already in the source. A stronger explanation names the visible cause:

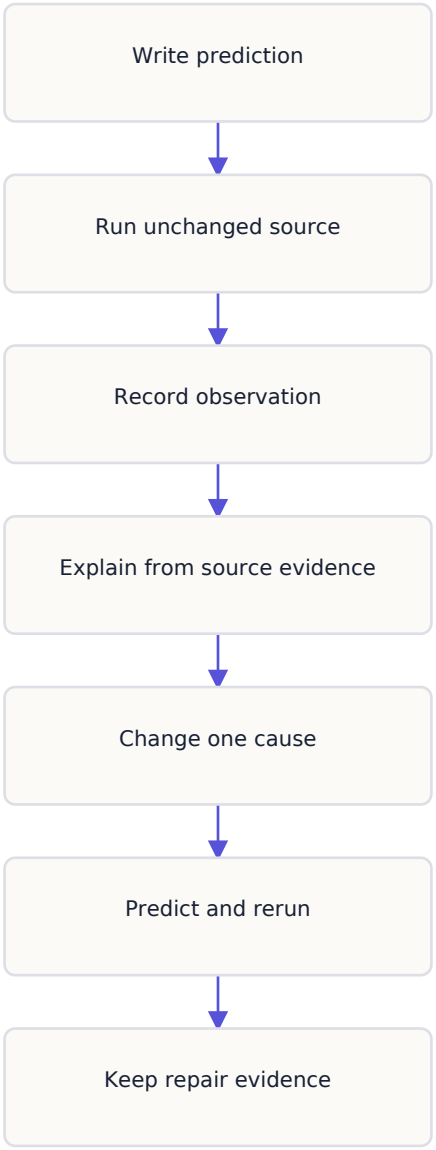
```text
The second print call contains the text Status: ready, so that call displayed
that text when Python reached it.
```

This line-by-line table slows the program down:

Line	Python reads	Visible result	State to carry forward
1	first `print(...)` call	`Python Learning Lab`	no hidden requirement for line 2
2	second `print(...)` call	`Status: ready`	line 1 remains displayed
3	third `print(...)` call	`Next step: predict before running`	program reaches its end

Later units use richer state tables for names, collections, branches, and loops. Here the key question is simple: which source line explains each output line?

A useful learning cycle preserves prediction and observation before revising the explanation.



Checkpoint: distinguish the records

Prediction and evidence checkpoint

Choose the best answer for each question.

1. Which prediction is most testable for a three-line program?

- **A.** It will display the three stated messages in source order with no extra line
- **B.** It will probably work somehow
- **C.** Python likes short programs
- **D.** The editor will solve it

2. Which sentence is an observation rather than an explanation?

- **A.** Python chose ready because it understood the project
- **B.** The run displayed Status: ready as its second output line

- **C.** The second call caused the result because calls always run
- **D.** The code is good

3. Which explanation is supported by the source print("Status: ready")?

- **A.** The computer inspected every installed tool
- **B.** The notebook guessed the learner’s intention
- **C.** The call contains the message that appears when Python reaches that line
- **D.** The output proves every future run will succeed

4. When should the prediction be preserved?

- **A.** After replacing it with the observed result
- **B.** Only after a successful run
- **C.** After several unrelated edits
- **D.** Before the run, so comparison remains honest

Write your choices before checking the answer key at the end of this book.

3. Change one cause and name every expected effect

A controlled experiment begins from a state you understand:

PYTHON
```
print("Python Learning Lab")
print("Status: ready")
print("Next step: predict before running")
```

Experiment A changes only line 2’s message:

PYTHON
```
print("Python Learning Lab")
print("Status: checked")
print("Next step: predict before running")
```

Record both change and non-change:

TEXT
```
Changed cause: ready became checked in source line 2.
Predicted effect: output line 2 becomes Status: checked.
Stable boundary: output lines 1 and 3 remain exactly unchanged.
Observed result: [copy all three lines after running]
Conclusion: [state whether the effect and stable boundary matched]
```

A stable boundary prevents selective checking. If line 2 is correct but line 1 vanishes, the experiment did not fully match your prediction.

Experiment B changes order rather than text:

PYTHON
```
print("Status: ready")
print("Python Learning Lab")
print("Next step: predict before running")
```

The changed cause is the position of the first two calls. The messages should remain the same; only their positions should exchange. This is a different hypothesis from changing message content.

Compare two experiments instead of mixing them

This version changes content and order together:

```python
print("Status: checked")
print("Learning Station")
print("Next action: open Unit 1")
```

It may be a useful final program, but it is a poor first experiment. Split it into stages:

1. change only `ready` to `checked`, then run;
2. change only the title, then run;
3. change only the final message, then run; and
4. reorder only if order is part of the requirement.

The intermediate runs cost a little time and save much more time when a result surprises you.

4. Treat surprise as a difference to investigate

Suppose you predicted three lines but observed only two:

```python
print("Python Learning Lab")
print("Status: ready")
```

Do not begin with “Python ignored line 3.” Inspect the current source first. There is no third instruction in this version. A useful record is:

```text
Prediction: three output lines.
Observation: two output lines.
First difference: the saved source currently contains only two print calls.
Hypothesis: line 3 was removed or not saved before the run.
One check: restore/save line 3, then rerun the same file.
```

Now consider a file that contains all three lines, but the terminal still shows an old two-line result. Possible hypotheses include:

- the file was edited but not saved;
- the terminal ran a different file with the same name;
- the current folder is not the expected folder; or
- the output shown is from an earlier run.

Do not choose all four repairs at once. Gather the cheapest discriminating evidence: save the file, read the exact run command, confirm the folder and file, then rerun. Lesson 5 practices those checks locally.

Use the earliest useful clue

This program cannot be read completely:

```python
print("Python Learning Lab")
print("Status: ready)
print("Next step: predict before running")
```

If Python reports a `SyntaxError` near line 2, do not first rewrite the complete line 3. Your current evidence points to the unclosed text on line 2. Repair one quote, rerun, and learn whether any later problem remains.

The repair:

```python
print("Python Learning Lab")
print("Status: ready")
print("Next step: predict before running")
```

A debugging claim should be allowed to fail. “The missing quote is the cause” becomes stronger only after replacing that quote lets the complete program run.

Checkpoint: design one useful experiment

Controlled experiment checkpoint

Choose the best answer for each question.

1. Only ready changes to checked in line 2. What should the experiment record?

- **A.** Line 2 should change and lines 1 and 3 should remain unchanged
- **B.** Every line should change
- **C.** Only that the program ran
- **D.** Nothing until the file is much larger

2. Why separate a message edit from a line reorder?

- **A.** Python cannot run two lines
- **B.** Separate runs make it clearer which cause produced each observed effect
- **C.** Every source line must be identical
- **D.** Reordering always creates a syntax error

3. A SyntaxError points near an unclosed quote on line 2. What is the best first experiment?

- **A.** Replace the entire program with unrelated code
- **B.** Edit line 3 before inspecting line 2
- **C.** Close the quote on line 2 and rerun the otherwise unchanged source
- **D.** Assume the interpreter installation is broken

4. The editor shows three calls but a rerun still displays an old two-line result. Which evidence should you check before rewriting code?

- **A.** How many tutorials exist online
- **B.** Whether another programming language is installed

- **C.** Whether every message is capitalized
- **D.** Whether the file was saved and the command ran that exact file from the expected folder

Write your choices before checking the answer key at the end of this book.

5. Build a learning log another person can follow

Create `learning-log.md` with one record per experiment. Use this structure:

```markdown
# Python Learning Lab log

## Experiment: change the status message

**Source before:** `print("Status: ready")`

**Prediction:** Only output line 2 will become `Status: checked`.

**Observed result:** Record all three output lines.

**Explanation:** Connect the changed source text to line 2 and confirm the
stable lines.

**Next controlled change:** State one change, but do not perform it in this
record.
```

A log is not a diary of every keystroke. It is a compact chain of reasoning. A future reader needs enough evidence to reproduce the run and understand why you chose the next step.

Mark uncertainty honestly

This is acceptable:

```text
Observation: SyntaxError points near line 2.
Hypothesis: the unclosed quote prevents Python from reading the program.
Confidence: likely, but not yet verified.
Check: close only the quote and rerun.
```

This overstates the evidence:

```text
Python is definitely broken because red text appeared.
```

An error color belongs to the interface, not the cause. The environment might use a different color, and a readable error message can still identify a simple source problem.

Record failures you repaired

Keeping only the final green result removes valuable information. Preserve one small failure:

```text
Failure: SyntaxError after I removed the closing quote on line 2.
Evidence: the message pointed near the unfinished text.
Hypothesis: the opening quote was unmatched.
Change: restored one closing quote.
Verified rerun: all three expected lines appeared in order.
```

This record is more useful than “fixed typo” because it contains evidence, hypothesis, change, and verification.

6. Ask for help without giving away the investigation

Before asking another person, a forum, or an AI assistant, prepare a minimal help packet:

```text
Goal: display the three Learning Lab status lines.
Environment: Colab code cell.
Smallest source: [paste the complete three lines]
Prediction: [exact lines]
Observation: [exact output or error]
Difference: [first mismatch]
Attempted check: [one change and rerun]
Question: [one bounded question]
```

A bounded question might be, “Which pair of delimiters is incomplete on line 2?” It is more useful than “Write the whole exercise for me.” Lesson 3 develops how to use the same packet with AI while protecting private information.

Know when to pause. If two controlled checks reject your hypotheses:

1. restore the last source whose behavior you understood;
2. reduce the example to the smallest failing lines;
3. copy the exact current evidence into the log;
4. reread the relevant section or primary documentation; and
5. ask a bounded question with the packet.

Repeated guessing is not persistence. Persistence means preserving evidence and choosing a better next experiment.

Checkpoint: decide what the evidence supports

Evidence and help checkpoint

Choose the best answer for each question.

1. Which failure record is most useful?

- **A.** Exact error evidence, smallest hypothesis, one repair, and verified rerun
- **B.** It was red and then it worked
- **C.** A final screenshot with no source
- **D.** Several guesses with no rerun

2. What should a bounded help packet include?

- **A.** Only the final question
- **B.** Goal, environment, smallest source, prediction, exact observation, attempted check, and focused question
- **C.** Every file on the computer
- **D.** Private credentials so the helper can log in

3. Two controlled checks reject your hypotheses. What is the strongest next move?

- **A.** Make many random edits
- **B.** Delete the evidence log
- **C.** Restore the last understood state, reduce the case, preserve evidence, and ask a bounded question
- **D.** Claim that Python is random

4. A repair has not been rerun yet. Which conclusion is honest?

- **A.** The problem is permanently solved
- **B.** Every future input will work
- **C.** The interpreter has been proven correct
- **D.** The change is a hypothesis until the rerun supplies evidence

Write your choices before checking the answer key at the end of this book.

7. Lab: write a complete evidence chain

Use this starting program:

```python
print("Python Learning Lab")
print("Status: ready")
print("Next step: predict before running")
```

Create `learning-log.md` and complete four linked records:

1. **Baseline:** predict and observe the exact three-line output.
2. **Controlled content change:** replace only `ready` with `checked`, predict the changed and stable lines, then observe them.
3. **Controlled order change:** restore `ready`, exchange only the first two source lines, predict the order, then observe it.
4. **Repair:** restore the baseline, remove only the closing parenthesis from line 3, run once, preserve the evidence, restore the delimiter, and verify the baseline again.

For each record, include:

- source or a precise source difference;
- prediction made before the run;
- complete observed output or error;
- explanation tied to the source;
- conclusion stating whether the prediction matched; and
- the next controlled step.

Hint: use stable boundaries

For each experiment, list what must remain unchanged before listing what should change. This makes partial success harder to mistake for complete success.

Show the shape of a strong solution

A controlled-change record can say:

```text
Source difference: line 2 contains checked instead of ready.
Prediction: line 2 becomes Status: checked; lines 1 and 3 remain exact.
Observation: [copy three lines]
Explanation: only the message supplied to the second print call changed.
Conclusion: matched / did not match, with the first difference.
Next step: restore ready before testing order.
```

For the repair, do not merely write “syntax error.” Copy the exception type and the portion that points near the incomplete third call. State that the missing) is a hypothesis, restore only it, and record the clean three-line rerun.

Key points

- A useful prediction names observable behavior precisely enough to be wrong.
- Observation records what happened; explanation claims why it happened.
- Preserve the original prediction instead of rewriting history after a run.
- A controlled experiment names one changed cause, its expected effect, and the boundaries that should stay stable.
- Debugging moves from evidence to a small hypothesis, one change, and a focused rerun.
- A strong help request includes the smallest reproducible source and exact evidence without unrelated or private material.

References

- The Python tutorial: using Python as a calculator and running simple statements
- Python errors and exceptions tutorial
- Jupyter documentation: running code

Use AI While Keeping the Thinking Yours

Online lesson and runnable notebook

Ask for bounded programming help, inspect a plausible response, verify every important claim locally, and protect private information.

- **Level:** Beginner
- **Estimated time:** 2–3 hours
- **You will learn:** Use AI for a bounded next step while keeping predictions, local execution, privacy decisions, and the final explanation under your control.
- **Practice in:** The supplied responses plus Google Colab, JupyterLab, or a local editor; no AI account is required

An AI assistant can produce fluent explanations and plausible code quickly. Fluency is not execution evidence. A response may misunderstand the goal, invent a function, assume a different software version, omit a boundary, or suggest a command that should not be trusted. The useful skill is not refusing all assistance or accepting all assistance. It is asking for a bounded next step and verifying the response against the program you actually have.

Questions to answer

- Which work should you attempt before asking an AI assistant?
- What context makes a programming question focused without exposing private material?
- How can you test a confident but incorrect answer?
- When should primary documentation take priority over generated prose?
- What assistance should you record in your learning log? :::

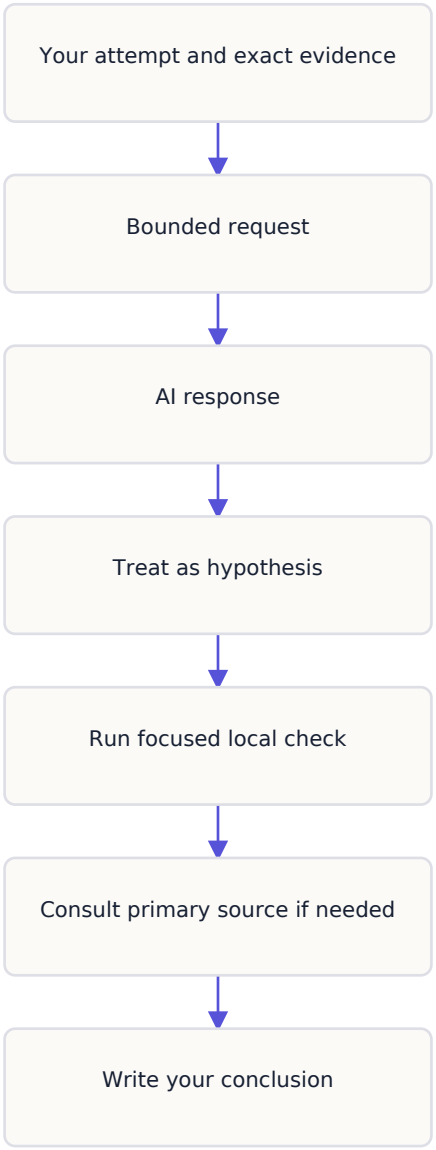
1. Keep the learning loop outside the chat window

Your evidence still belongs in the Python Learning Lab:

1. read the current source;
2. write your own prediction;
3. run the smallest relevant example;
4. preserve the exact observation or error;
5. identify the first difference;
6. ask for one kind of assistance if needed;
7. treat the answer as a hypothesis;
8. run a focused check; and
9. write the conclusion in your own words.

If the assistant supplies the prediction, the run, the explanation, and the final answer before you attempt them, it has completed the part that was meant to exercise your reasoning. A useful request creates a next step without hiding your responsibility for the result.

The assistant can inform a hypothesis, but verified evidence must return to your local learning loop.



Consider the first program:

```python
print("Python Learning Lab")
print("Status: ready")
print("Next step: predict before running")
```

Before asking anything, you can already predict three lines, run them, and map each result to a source line. If a closing quote is missing, you can compare the line with a complete neighbor. Ask for help when the evidence leaves a focused question, not simply because a chat box is available.

2. Ask for one useful kind of help

Four request shapes work well in this course.

Ask for a hint

```text
I am learning to pair delimiters. This Python line raises SyntaxError:
print("Status: ready)
Do not give me a complete replacement program. Point to the first unmatched
opening symbol and ask me what should close it.
```

The request states the goal, supplies the smallest source and exact exception type, and limits the answer.

Ask for an explanation to review

```text
I predicted three output lines for the source below, and the prediction matched.
Explain which source line produces each output line. Keep the explanation to
three numbered sentences so I can compare it with my own trace.
```

Write your trace first. Then compare claims rather than copying the response.

Ask for a counterexample

```text
My claim is: if a Python program runs without SyntaxError, its output must meet
the requirement. Give one two-line counterexample using only print calls. Do not
explain it until after the code block.
```

This asks the assistant to challenge a claim. You still run the counterexample and explain why it matters.

Ask for a review of your reasoning

```text
Here are my prediction, observed output, and explanation. Identify one sentence
that is not supported by the source or output. Ask a question instead of
rewriting the explanation for me.
```

A review request can reveal an unsupported leap while leaving the revision to you.

A vague request hides the next step

```text
Teach me Python and fix my code.
```

This does not identify the code, expected behavior, evidence, or boundary. A long answer may still fail to address your actual obstacle. Narrow the request to one discrepancy.

Checkpoint: choose a bounded request

Bounded assistance checkpoint

Choose the best answer for each question.

1. Which request best preserves the learner’s work on an unmatched quote?

- **A.** Point to the first unmatched delimiter and ask me what should close it; do not replace the whole program
- **B.** Complete every course exercise for me
- **C.** Return only a final file with no explanation
- **D.** Assume any red output means Python is broken

2. Which context makes a programming question focused?

- **A.** Every file and credential on the device
- **B.** Goal, smallest relevant source, exact evidence, prediction, attempted check, and one question
- **C.** Only the words it does not work
- **D.** A screenshot with the important text cropped out

3. When should you normally write your prediction?

- **A.** After copying the assistant’s final answer
- **B.** Only when the assistant refuses
- **C.** Before requesting the assistant’s prediction or explanation
- **D.** After deleting the observed result

4. Why is Teach me Python and fix my code a weak debugging request?

- **A.** It contains too much exact output
- **B.** It reveals a precise controlled experiment
- **C.** It forbids a complete solution
- **D.** It omits the specific goal, source, evidence, and discrepancy

Write your choices before checking the answer key at the end of this book.

3. Test a plausible response instead of admiring it

You do not need a live service for this practice. Review this supplied response:

Replace `print` with `echo`; `echo` is the standard Python command for showing text, and the output will remain the same.

The response is confident and wrong for ordinary Python. Turn it into a prediction before running:

```
PYTHON
echo("Python Learning Lab")
```

The source has balanced punctuation, so Python can begin running it. In a normal fresh Python environment, however, there is no built-in name `echo`; the run raises `NameError`. The original built-in used in this course is `print`:

```
PYTHON
print("Python Learning Lab")
```

Your verification record can say:

```text
AI claim: echo is the standard Python display command.
Prediction from the claim: echo("Python Learning Lab") displays the title.
Observed evidence: NameError says echo is not defined in a fresh environment.
Primary/local check: print("Python Learning Lab") displays the title.
Conclusion: reject the claim; keep print for this program.
```

Do not merely say “the AI hallucinated.” Identify the exact claim, the test that could disprove it, and the evidence that did.

A response can be partly right

Review a second supplied response:

The three `print` calls display three lines. Because computers can execute quickly, the order of those lines is not guaranteed.

The first sentence fits the source. The second conflicts with the ordinary sequential behavior of this script. Run the same program several times:

```python
print("one")
print("two")
print("three")
```

Repeated matching runs are useful evidence for this example, though they do not prove every possible program is sequential in the same simple way. Rewrite only the unsupported sentence:

```text
For this script, Python reaches the three print calls from top to bottom, so the
observed output is one, two, three in that order.
```

A mixed response should be repaired claim by claim. Rejecting everything can throw away a useful clue; accepting everything imports the unsupported part.

A working answer can solve the wrong requirement

Suppose the requirement is exactly:

```text
Status: ready
```

An assistant suggests:

```python
print("The Python Learning Lab is ready to begin")
```

This source runs, and the general meaning is similar, but the exact output does not meet the contract. Verification is against the stated requirement, not against whether the answer looks polished.

4. Prefer primary sources for version-sensitive claims

Generated prose may mix older and newer interfaces. When a claim concerns an installation command, supported Python version, editor menu, notebook behavior, security setting, or library API:

1. identify the exact claim;
2. note the software and version in your environment;
3. open the official documentation for that software;
4. find the page that directly owns the behavior;
5. compare it with a small local check; and
6. record any remaining uncertainty.

Do not ask an assistant to fabricate a citation and then trust the link title. Open the page and confirm that it contains the relevant instruction. Search results, forum snippets, and generated summaries can help locate a source; they are not substitutes for reading it.

For Python's `print`, the built-in functions documentation is the owning reference. Unit 0 does not require you to understand the complete signature. You only need to confirm that `print` is a Python built-in and compare that fact with the local run.

Checkpoint: verify claims with evidence

AI claim verification checkpoint

Choose the best answer for each question.

1. An answer claims echo("Ready") is the ordinary Python replacement for print. What is the best first verification?

- **A.** Predict the claim's effect, run the smallest echo call in a fresh Python environment, and preserve the exact result
- **B.** Repeat the claim more confidently
- **C.** Change the whole project before running it
- **D.** Assume a code block proves execution

2. A response has one supported sentence and one unsupported sentence. What should you do?

- **A.** Accept both because one is right
- **B.** Evaluate and revise each claim against source, output, and the requirement
- **C.** Reject all future assistance
- **D.** Hide the unsupported sentence from the log

3. An assistant gives a version-sensitive installation instruction. Which evidence is strongest?

- **A.** The answer uses a confident tone
- **B.** The command appears in many generated paragraphs
- **C.** Current official documentation for the identified tool/version plus a focused local check
- **D.** An unrelated tutorial screenshot

4. Suggested code runs, but its output differs from the exact required line. What has been verified?

- **A.** The requirement is unnecessary
- **B.** Every future case is correct
- **C.** The assistant understood the goal perfectly
- **D.** Only that the source ran; it still fails the stated output contract

Write your choices before checking the answer key at the end of this book.

5. Protect information before sending a request

Treat text sent to an external service as information leaving your current workspace. Before pasting, ask:

- Does this contain a password, API key, recovery code, private URL, or session token?
- Does it contain names, addresses, health information, grades, or other personal records?
- Does a school, employer, research agreement, or client prohibit sharing this code or data?
- Does it reveal the answer to an assessment that you are expected to complete yourself?
- Can I replace the real material with a tiny invented example that preserves the technical problem?

Never “anonymize” a secret by changing only one character. Remove it completely and rotate a real credential if it was exposed. Do not paste `.env` files, authentication headers, private keys, or full proprietary repositories.

Reduce a private case to an invented one

Unsafe request:

```text
Here is our private customer export, internal server address, and access token.
Why does this script fail?
```

Safer request:

```text
I replaced the private data with these three invented lines and removed all
credentials. My tiny Python example should display three labels but raises this
exact error: [...]. Which source delimiter should I inspect first?
```

The reduced case is often easier to debug because it removes unrelated details.

Treat commands as proposals, not instructions

An assistant may suggest a shell command that downloads and immediately runs unreviewed code, deletes files, changes permissions, or installs software outside your intended environment. Do not run a command merely because it is formatted as code.

Before any unfamiliar command:

1. identify what executable it invokes;
2. read each argument and path;
3. determine which files, network services, or permissions it affects;
4. compare it with official installation or tool documentation;

5 remove pipelines that hide downloaded content before execution;

6 ask a trusted maintainer or administrator when the effect is unclear; and

7 keep a backup or disposable environment when a legitimate operation is destructive.

Unit 0 never requires disabling security controls. A managed device restriction is evidence to record and escalate, not an obstacle to bypass.

6. Record assistance without surrendering authorship

A useful assistance record is short:

```
MARKDOWN
## Assistance record

- **Question asked:** Identify the first unmatched delimiter; do not rewrite the program.
- **Suggestion received:** The text opened after `print(` has no closing quote.
- **Verification:** Added only the quote, reran the original three lines, and observed the required out ↪
    put.
- **My conclusion:** The repair worked because the opening and closing text delimiters now form a compl ↪
    ete source instruction.
```

Record assistance when it materially changes your reasoning, source, or next step. Follow any school, employer, publication, or project disclosure policy that applies. A record does not turn unverified code into your understanding; you must still be able to explain and maintain it.

A copied explanation can be detected by a simple test: close the response, look at your source and output, and explain the connection without its wording. If you cannot, ask for a smaller hint or reconstruct the evidence.

Checkpoint: protect and disclose

Privacy and ownership checkpoint

Choose the best answer for each question.

1. A failure occurs while processing a private customer file. What is the safest useful request?

- **A.** Remove credentials and private data, create a tiny invented case with the same technical shape, and share exact non-sensitive evidence
- **B.** Paste the full export and access token
- **C.** Send the entire private repository
- **D.** Publish the customer file before asking

2. An assistant proposes an unfamiliar command that downloads and executes a script. What should you do first?

- **A.** Run it because it is in a code block
- **B.** Inspect the executable, arguments, paths, permissions, and official documentation before deciding
- **C.** Add administrator privileges automatically
- **D.** Disable security controls

3. Which assistance record best preserves ownership of the work?

- **A.** AI fixed it
- **B.** A pasted answer with no run
- **C.** Question, suggestion, focused verification, observed result, and learner's own conclusion
- **D.** No record even when policy requires disclosure

4. A learner does not have or want an AI account. How can they complete this lesson?

- **A.** They cannot learn Python
- **B.** They must share someone else's account
- **C.** They should skip verification
- **D.** They can analyze the supplied responses and run the same focused local checks

Write your choices before checking the answer key at the end of this book.

7. Lab: audit a confident answer

Use this supplied answer; do not consult a live service unless you choose to:

Your script should use `echo` because Python 3 replaced `print`. Change all three calls at once. If the code still fails, install a different Python and paste your environment file into the chat so I can inspect it.

Complete an assistance entry in `learning-log.md`:

1. Break the response into at least four separate claims or instructions.
2. Mark which claims concern Python behavior, environment changes, and private information.
3. Predict the result of the smallest `echo("Python Learning Lab")` check.
4. Run it in a fresh cell or from a clean script and copy the exact result.
5. Run the documented `print("Python Learning Lab")` comparison.
6. Use the Python built-in functions documentation to identify the supported display function.
7. Explain why changing all calls and the interpreter together would be a weak experiment.
8. Explain why an environment or credential file should not be pasted.
9. Write a safer follow-up request that asks for one hint without the final program.
10. Finish with your own two-sentence conclusion.

Hint: separate behavior from safety

Test the `echo` claim with one line. Review the "change all three," "install a different Python," and "paste your environment file" instructions separately; they require different evidence and carry different risks.

Show one possible audit path

A fresh ordinary Python run of:

```
PYTHON
echo("Python Learning Lab")
```

raises `NameError` because `echo` has not been defined. This comparison:

```python
print("Python Learning Lab")
```

displays the required title, and Python's built-in functions reference owns the `print` behavior. Reject the unsupported replacement claim. Test one call before three; do not replace the interpreter without evidence; do not share environment files until you have inspected them and removed secrets. A safer request is, "Using the exact `NameError` below, point me to the Python built-in I should verify in the official reference; do not rewrite the complete script."

Key points

- Make your own attempt and prediction before requesting generated reasoning.
- Ask for one bounded kind of help: a hint, explanation to review, counterexample, or critique of your reasoning.
- Treat every important response claim as a hypothesis until code, output, the requirement, and primary documentation support it.
- A program can run successfully and still solve the wrong task.
- Remove secrets, personal data, private code, and assessment answers before sending anything to an external service.
- Inspect unfamiliar commands and never bypass security controls just because a response suggests it.
- Record material assistance, local verification, and your own conclusion.

References

- Python built-in functions reference: `print`
- NIST Generative Artificial Intelligence Profile
- OWASP guidance for generative AI and large language model applications

Work Reliably in Colab and Jupyter

Online lesson and runnable notebook

Build a readable practice notebook, diagnose stale output and hidden runtime state, then prove the notebook works from a clean top-to-bottom run.

- **Level:** Beginner
- **Estimated time:** 2.5–4 hours
- **You will learn:** Keep the saved notebook, live Python runtime, cell execution order, and displayed output consistent enough for a clean rerun.
- **Practice in:** Google Colab or JupyterLab

A notebook places explanation, code, and output close together. That makes it a friendly learning environment—and makes it possible to create a convincing page whose visible cells do not reproduce the visible result. This lesson builds a Python Learning Lab notebook that another person can restart and run from top to bottom.

Questions to answer

- What is saved in a notebook document, and what exists only in the live runtime?
- Why can cell order differ from the order shown on the page?
- How can displayed output become stale?
- What does restart-and-run-all actually test?
- Which notebook files and information should you save, download, or avoid sharing? :::

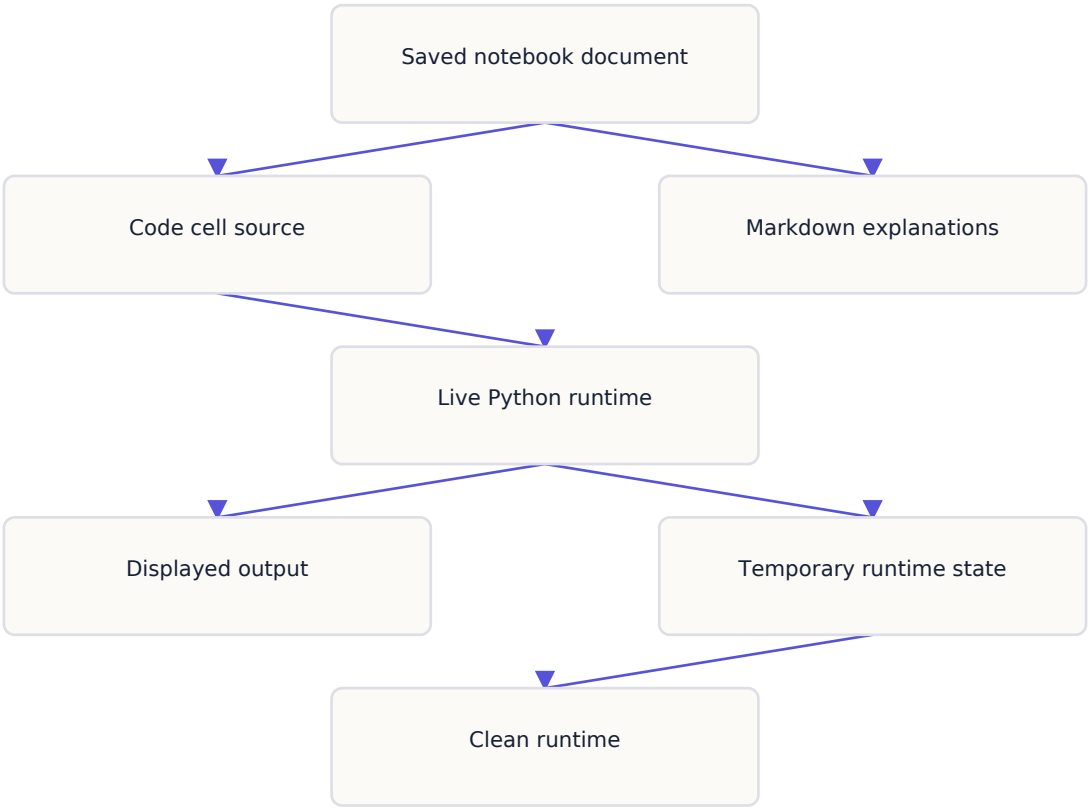
1. Separate the document from the runtime

A notebook interface brings several things together:

Part	What it is	What can happen to it
Notebook document	Saved cells, their order, metadata, and possibly stored output	Can be renamed, copied, downloaded, or shared
Code cell	Editable Python source inside the document	Can be changed without being run
Markdown cell	Formatted explanation, headings, links, and predictions	Does not execute as Python
Runtime or kernel	The live Python process that executes code	Can disconnect, restart, or remember hidden state
Displayed output	A record shown beneath a cell	Can remain visible after source changes
Runtime file	A file created on the current remote or local machine	May disappear when a hosted runtime resets

The notebook page and the Python process are related but not identical. Saving a notebook does not keep a hosted runtime alive forever. Restarting a runtime does not normally delete the saved source cells. Editing a cell does not automatically rerun it.

The saved notebook supplies cells to a temporary runtime, which returns output that the page displays.



Temporary runtime state → Clean runtime: lost on restart

In Google Colab, make or save a copy you control before treating a shared notebook as your work record. In JupyterLab, confirm the file is inside the folder you intend to preserve. Give it the clear name `first_notebook.ipynb`.

2. Build a notebook that explains its own run

Use four cells.

Cell 1 — Markdown

```
MARKDOWN
# Python Learning Lab

Prediction: the next code cell will display three status lines in source order.
```

Cell 2 — Python

```
PYTHON
print("Python Learning Lab")
print("Status: ready")
```

```
PYTHON (CONTINUED)
print("Next step: predict before running")
```

Cell 3 — Markdown

```
MARKDOWN
## Observation

Record all three observed lines and explain which source call produced each one.
```

Cell 4 — Python experiment

```
PYTHON
print("Python Learning Lab")
print("Status: checked")
print("Next step: predict before running")
```

The notebook should tell a future reader what question the code answers. A page of unlabeled code and old output is not yet a learning record.

Markdown is part of the artifact

Use Markdown headings to expose structure rather than decorating every sentence. A useful notebook can be scanned for:

- goal;
- prediction;
- original source;
- observed output;
- controlled experiment;
- repair evidence; and
- conclusion.

Do not place Python source inside a Markdown cell and expect it to run. Do not place a prose heading inside a code cell and expect Python to treat it as page structure.

Checkpoint: identify notebook parts

Notebook parts checkpoint

Choose the best answer for each question.

1. Which statement correctly distinguishes a notebook document from its runtime?

- **A.** The document stores cells; the runtime is the live Python process that executes code
- **B.** They are always the same permanent object
- **C.** The runtime is only a Markdown heading
- **D.** The document disappears after every code cell

2. What is a strong use for a Markdown cell?

- **A.** Running an unlabeled Python instruction

- **B.** Recording a heading, prediction, observation, and explanation
- **C.** Selecting the Python executable
- **D.** Installing every local tool

3. You edit a code cell but do not run it. What happens to the old displayed output?

- **A.** It proves the new source is correct
- **B.** It must vanish in every notebook system
- **C.** It may remain visible even though it belongs to an earlier version of the source
- **D.** It automatically updates to the predicted output

4. A file exists only on a temporary hosted runtime. What should you assume?

- **A.** It is permanently stored in every account
- **B.** It is automatically committed to Git
- **C.** It can never be downloaded
- **D.** It may disappear when the runtime resets unless you deliberately preserve it

Write your choices before checking the answer key at the end of this book.

3. Cell position is not execution history

A notebook lets you run cell 4 before cell 2. The page order remains 1, 2, 3, 4, but runtime history follows the order in which you pressed Run.

This pair previews **assignment**, which Unit 1 teaches formally. For this notebook diagnosis, understand only that the first cell creates the name `status` in the current runtime and the second tries to display its current value.

Visible Cell A

```python
status = "ready"
```

Visible Cell B

```python
print(status)
```

A fresh runtime that runs B before A raises `NameError` because `status` has not been created there. Run A and then B; the output becomes `ready`.

Now add a third cell later on the page:

```python
status = "checked"
```

If you run the third cell and then return to run B, B displays `checked` even though the visible assignment above it still says `ready`. The result depends on runtime history that the top-to-bottom page does not reveal.

Use a trace table:

Action order	Cell run	Runtime state after the action	Output
1	`status = "ready"`	`status` refers to ready	none
2	`status = "checked"`	`status` now refers to checked	none
3	`print(status)`	still checked	`checked`

The notebook is not random. It followed the actual execution order. The page is misleading because that history is not the order a fresh reader will use.

A clean top-to-bottom trace

Restart the runtime. Now run:

```python
status = "ready"
```

then:

```python
print(status)
```

then:

```python
status = "checked"
```

then:

```python
print(status)
```

Expected output sequence:

```text
ready
checked
```

Every dependency now appears before the cell that uses it.

4. Stale output can tell a false story

Suppose a cell originally contains:

```python
print("Status: ready")
```

You run it and see:

```text
Status: ready
```

Then you edit the source to:

```python
print("Status: checked")
```

but do not rerun. Some notebook interfaces keep the old `Status: ready` output visible. A screenshot would show new source beside old evidence.

Before trusting a notebook:

1. mark any edited-but-not-run cell;
2. run the changed cell and inspect new output;
3. restart the runtime;
4. run every code cell from the top in page order;
5. confirm no cell fails or depends on a value created later; and
6. save the notebook after the clean run.

A stored output is a useful record only when you know which source and runtime state produced it.

Checkpoint: reconstruct execution history

Notebook state checkpoint

Choose the best answer for each question.

1. A fresh runtime runs print(status) before any cell creates status. What should you expect?

- **A.** A NameError because the name does not yet exist in that runtime
- **B.** The notebook guesses ready
- **C.** Every previous session is restored automatically
- **D.** The cell becomes Markdown

2. A later cell changes status to checked, then an earlier print(status) cell is rerun. What will that print use?

- **A.** The nearest visible text regardless of history
- **B.** The current runtime value checked
- **C.** No value because earlier cells cannot rerun
- **D.** Always ready

3. Source now says print("Status: checked") but visible output still says Status: ready. What is the best first explanation to test?

- **A.** Python changes text randomly
- **B.** Quotation marks reverse the word
- **C.** The cell was edited after its last run, so the visible output may be stale
- **D.** The notebook cannot display strings

4. Why can a notebook work for its author but fail top-to-bottom for a new reader?

- **A.** New readers cannot run Python
- **B.** Markdown deletes all code
- **C.** Saved notebooks never contain cells
- **D.** The author’s runtime may contain values created by a hidden out-of-order history

Write your choices before checking the answer key at the end of this book.

5. Restart and run all is an experiment

“Restart and run all” is not a ritual that repairs every notebook. It tests a specific claim:

The visible notebook contains every required step in an order that succeeds from an empty runtime.

The experiment has a baseline and observable result:

1. save your current source cells;
2. restart or disconnect and reconnect to obtain an empty runtime;
3. run cells from the first code cell to the last without manually jumping ahead;
4. stop at the first error;
5. record the cell, exception, and missing dependency;
6. repair the notebook’s visible order or source;
7. restart again; and
8. repeat until a clean run succeeds.

If you repair the runtime by manually running a later setup cell first, the page still fails its clean-run contract. Move or rewrite the dependency so page order works.

Read execution indicators carefully

Many notebook interfaces show an execution count or running indicator beside a code cell. Counts can reveal that lower cells ran earlier, but do not rely on one interface’s exact appearance. The strongest test is a restart and complete run, not a tidy-looking sequence of old numbers.

Interrupt work that will not finish

A cell can remain busy because it is doing a large task or because the code does not stop. Use the interface’s interrupt or stop action rather than refreshing repeatedly. Preserve the source and note which cell was running. Unit 4 teaches loop termination; Unit 8 teaches systematic debugging. Unit 0 asks only that you know the runtime can be interrupted and restarted.

Do not use this deliberately endless example as normal practice; read its shape only:

```text
while True:
    keep_working_forever
```

The text fence prevents the generated notebook from running it. Later you will learn what a loop condition means.

6. Save the right artifact and share it carefully

A hosted runtime may disappear. Preserve:

- the notebook document in a durable location you control;
- important input files or instructions for recreating them;
- the environment or package information when external packages become relevant; and
- the clean-run evidence in Markdown.

Do not assume a notebook saved in a shared course location belongs to your account. Make your own copy or download it. Reopen that copy before deleting the original session.

Before sharing, inspect:

- Markdown cells for names, email addresses, class identifiers, or private notes;
- source cells for passwords, tokens, paths containing personal names, or private URLs;
- output cells for data that the cleaned source no longer shows; and
- notebook metadata and attached files when the context is sensitive.

Deleting a secret from the current source is not enough if an old output cell still displays it. Rotate any real credential that was exposed.

Choose a notebook, script, or published page for its job

Artifact	Best fit now	Important boundary
Colab/Jupyter notebook	Explore, predict, explain, and display nearby output	Runtime state can be hidden and hosted files can be temporary
`.py` script	Run saved source from top to bottom	Rich prose and old output do not live beside the source
Quarto page	Publish structured teaching or documentation	Rendering is not the same as the learner’s live runtime

Unit 10 moves reusable behavior into modules and project structure. Unit 15 builds published documentation. Here, use a notebook as an honest lab record.

Checkpoint: prove a clean notebook

Clean notebook checkpoint

Choose the best answer for each question.

1. What claim does restart and run all test?

- **A.** The visible notebook contains the required steps in a successful top-to-bottom order
- **B.** Every possible input is correct
- **C.** The runtime can never disconnect
- **D.** The notebook has no Markdown

2. A clean run fails at the third code cell. What should you preserve first?

- **A.** Only the final cell
- **B.** The failing cell, exact exception, and dependency it expected from earlier visible cells
- **C.** A different notebook’s output
- **D.** Several unrelated edits

3. A secret was removed from source, but an old output cell still shows it. What is true?

- **A.** The notebook is safe because source changed
- **B.** Old output cannot be shared
- **C.** The notebook still exposes the secret; remove the output and rotate the real credential
- **D.** Restarting guarantees nobody saw it

4. A cell does not finish. What is the safest first runtime action?

- **A.** Run many more copies of the cell
- **B.** Install another Python immediately
- **C.** Assume the device is permanently broken
- **D.** Use the notebook’s interrupt/stop action, preserve the cell, and inspect it before rerunning

Write your choices before checking the answer key at the end of this book.

7. Lab: prove `first_notebook.ipynb` from a clean runtime

Create or rename a notebook to `first_notebook.ipynb`. Build these sections:

1. `# Python Learning Lab`
2. a Markdown prediction for the exact three-line baseline;
3. one baseline code cell with the three `print(...)` calls;
4. a Markdown observation that maps source lines to output lines;
5. a controlled experiment cell that changes only `ready` to `checked`;
6. a Markdown comparison naming the changed and stable output;
7. two state cells: one creates `status = "ready"`, and the next displays it;
8. a Markdown note explaining that assignment is a Unit 1 preview; and
9. a final clean-run record.

Then create one hidden-state failure deliberately:

1. run the state-creation cell;
2. change the live value in a temporary lower cell to `checked`;
3. rerun the display cell and observe the surprising result;
4. delete the temporary cell;
5. restart the runtime; and
6. run all visible cells from the top.

Completion evidence:

```text
Notebook filename:
Clean-run action used:
First code cell result:
Controlled experiment result:
State display result after restart:
Any warning or failure and its repair:
Durable save/download location:
```

Hint: repair the page, not only the runtime

After exposing hidden state, remove the temporary mutation and make sure every visible use has its required creation above it. Then restart again; do not rely on manually running a cell out of order.

Show one possible notebook order

Use this sequence:

1. Markdown title and prediction.
2. Baseline program:

```python
print("Python Learning Lab")
print("Status: ready")
print("Next step: predict before running")
```

3. Markdown observation.
4. Controlled experiment:

```python
print("Python Learning Lab")
print("Status: checked")
print("Next step: predict before running")
```

5. Markdown comparison.
6. State preview:

```python
status = "ready"
```

7. Display:

```python
print(status)
```

8. Markdown clean-run record.

Restart and run all. The baseline and controlled experiment each display their three expected lines; the final display is `ready`. Save after this clean run.

Key points

- The notebook document, live runtime, displayed output, and temporary files have different lifecycles.

- Page position does not prove execution order.
- Editing source does not refresh old output.
- Restart-and-run-all tests whether visible dependencies succeed from an empty runtime in page order.
- Repair the notebook’s source or order instead of manually recreating hidden state.
- Save important notebooks and files deliberately, and inspect source, output, metadata, and attachments before sharing.

References

- Project Jupyter documentation: running notebooks and kernels
- JupyterLab user guide: notebooks
- Google Colab FAQ
- CommonMark Markdown reference

Build a Dependable Local Python Workspace

Online lesson and runnable notebook

Verify one Python launcher, run a saved script from a known folder, align VS Code with that interpreter, and confirm a minimal isolated environment.

- **Level:** Beginner
- **Estimated time:** 3–5 hours; installation time varies by device
- **You will learn:** Build and diagnose one local path from project folder to script, interpreter, editor, virtual environment, and optional local notebook.
- **Practice in:** A terminal and VS Code on Windows, macOS, or Linux, with Colab as the hosted fallback for restricted devices

A local Python workspace lets you keep source files in folders, run them from a terminal, connect an editor to a known interpreter, and isolate tools for one project. The goal is not to collect software. It is to create one path you can verify from beginning to end.

Installation screens and commands evolve. This lesson uses current official Python, VS Code, and Jupyter guidance and teaches evidence that remains useful when an interface changes.

Questions to answer

- Which Python launcher does your terminal actually run?
- How do the current folder and script path determine which file runs?
- Why can VS Code and the terminal disagree about an interpreter?
- What does a virtual environment isolate, and how can you verify it without trusting a prompt label?
- What should you record when installation is blocked by a managed device? :::

1. Choose a route your device permits

Use the local route when you can install or already have a supported Python and can create files. Use the hosted route when a managed school, work, library, or mobile device prevents local installation. Both routes complete the lesson by producing exact environment evidence.

Evidence	Local route	Hosted/restricted route
Python runtime	verified terminal launcher	Colab runtime information
Learning Lab	local folder and `first_run.py`	saved Colab notebook and downloaded script copy
Clean run	command plus exact output	restart/run-all plus exact output
Constraint	any warning or mismatch	installation restriction and who can authorize access

Evidence	Local route	Hosted/restricted route
Fallback	Colab link or saved notebook	durable copy/download and future local-access plan

Do not bypass administrator policy, disable antivirus, change execution policy, or weaken certificate checks for this course. Record the restriction and ask the device owner or administrator. Later project units require filesystem and terminal practice, so include a concrete plan rather than silently assuming the hosted route solves every future task.

2. Install Python from an owning source

If Python is already available, verify it before installing another copy. If it is not available, use the current official instructions for your platform:

- **Windows:** Python's current documentation directs most learners to the Python Install Manager available through Python's downloads page or the Microsoft Store. Older tutorials may describe the legacy launcher or an old installer screen; prefer the current official Windows page.
- **macOS:** Python.org provides signed installer packages and platform-specific completion steps. Do not remove or modify the Apple-controlled system Python.
- **Linux:** distribution package managers commonly own system Python. Follow the Python Unix guidance and your distribution's documentation rather than replacing system-managed files manually.

This repository declares Python 3.10 or newer for its helper package. Record the version you actually have. A version is evidence, not a command to upgrade every working system immediately.

Find one working launcher

Open a new terminal after installation. Try only the commands appropriate to your platform until one works:

Platform	First checks
Windows PowerShell or Command Prompt	`python --version`, then `py --version` if needed
macOS terminal	`python3 --version`
Linux terminal	`python3 --version`, or the launcher documented by the distribution

A successful result resembles:

```text
Python 3.13.5
```

Your exact supported version may differ. Record the launcher—`python`, `py`, or `python3`—as **your Python command**. Use that same command in the following checks. Do not randomly alternate among them.

For example, if your command is `python3`:

```bash
python3 -c "print('Python Learning Lab')"
python3 -m pip --version
```

If your command is `python`:

```bash
python -c "print('Python Learning Lab')"
python -m pip --version
```

On current Windows installations, this may be:

```powershell
python -c "print('Python Learning Lab')"
python -m pip --version
```

The first command proves that the launcher can execute a one-line program. The second asks that same interpreter to run its package installer. It is stronger than invoking a bare `pip`, which might belong to another installation.

Preserve the executable path

Run one form with your chosen launcher:

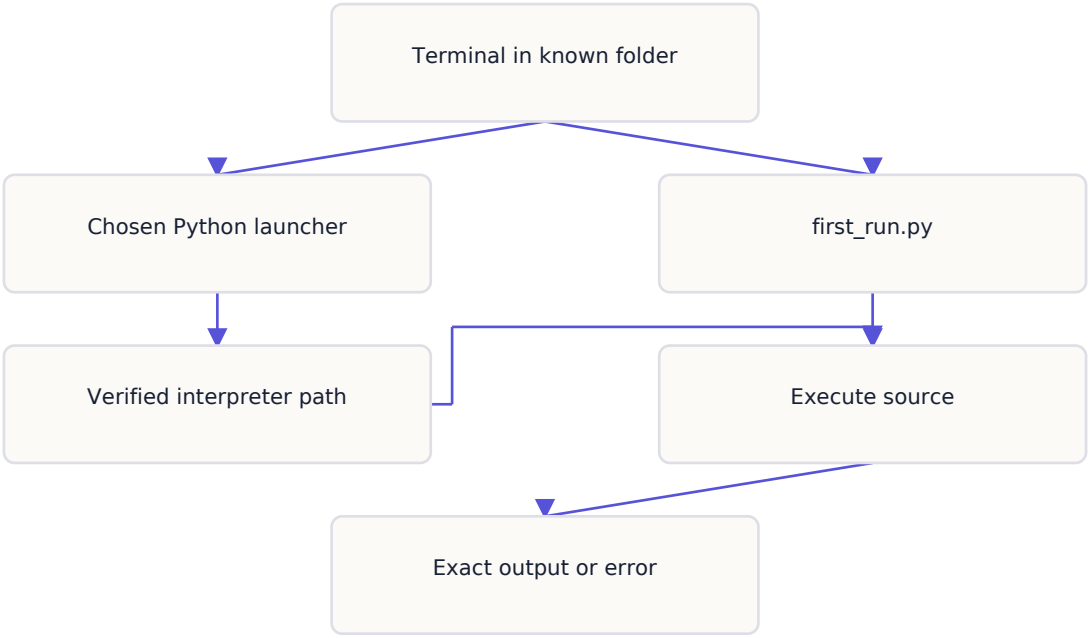
```bash
python -c "import sys; print(sys.executable)"
```

or:

```bash
python3 -c "import sys; print(sys.executable)"
```

This line previews `import` and the standard-library `sys` module, taught in Unit 10. You do not need to explain those mechanisms yet. Preserve the path because it identifies the executable that produced your evidence.

A dependable run connects the terminal command to one interpreter, one source file, and observed output.



Checkpoint: verify the launcher

Python launcher checkpoint

Choose the best answer for each question.

1. python3 –version works on a learner’s computer. What is the best next habit?

- **A.** Record python3 as the working launcher and use it consistently in the next checks
- **B.** Install three more Python distributions immediately
- **C.** Alternate randomly among python, py, and python3
- **D.** Assume every editor already uses it

2. Why prefer the chosen launcher followed by -m pip for this setup?

- **A.** It makes every package safe
- **B.** It asks that identified Python interpreter to run its pip module
- **C.** It disables virtual environments
- **D.** It turns a script into Markdown

3. What useful evidence does print(sys.executable) provide here?

- **A.** The complete Python language specification
- **B.** Every package ever installed
- **C.** The filesystem path of the interpreter running the check
- **D.** A permanent guarantee about VS Code

4. A managed device blocks Python installation. What should the learner do?

- **A.** Disable its security controls

- **B.** Use an unknown administrator password
- **C.** Delete system files
- **D.** Record the restriction, continue through the hosted route, and identify the authorized support needed later

Write your choices before checking the answer key at the end of this book.

3. Know which folder the terminal is using

A terminal command operates from a **current working directory**. Before running a relative filename such as `first_run.py`, confirm that the terminal is in the folder containing it.

macOS and Linux shell

Show the current folder and list its entries:

```bash
pwd
ls
```

Create the Learning Lab if needed, move into it, and list it:

```bash
mkdir python-learning-lab
cd python-learning-lab
pwd
ls
```

Windows PowerShell

Use PowerShell's explicit commands:

```powershell
Get-Location
Get-ChildItem
New-Item -ItemType Directory -Name python-learning-lab
Set-Location python-learning-lab
Get-Location
Get-ChildItem
```

PowerShell also provides familiar aliases such as `cd` and `dir`, but the explicit names make the first record easier to explain.

Do not type both platform branches. Choose the one that matches your terminal. Record the final folder path.

Save the script in that folder

Create `first_run.py` with a plain-text editor:

```python
print("Python Learning Lab")
print("Status: ready")
print("Next step: predict before running")
```

Confirm the filename ends with `.py`, not `.py.txt`. Some file managers hide known extensions, so inspect the editor tab and terminal listing.

Run it with your chosen launcher. Examples:

```bash
python3 first_run.py
```

```bash
python first_run.py
```

```powershell
python first_run.py
```

Expected output:

```text
Python Learning Lab
Status: ready
Next step: predict before running
```

Diagnose a file-not-found boundary

Suppose the terminal says it cannot open `first_run.py`. Do not reinstall Python first: Python ran far enough to report that the supplied file path was not found. Check:

1. the current folder path;
2. the entries listed in that folder;
3. the exact filename and extension;
4. spelling and capitalization; and
5. the run command.

A useful record:

```text
Command: python first_run.py
Evidence: Python reports that first_run.py cannot be opened.
Current folder: [copy Get-Location or pwd result]
Listed files: [copy the relevant names]
First hypothesis: the file is saved in a different folder / named first_run.py.txt.
One check: move to the containing folder or correct the filename, then rerun.
```

The fact that Python produced a file error is evidence that the launcher exists. Do not replace a working interpreter while the current evidence concerns a path.

4. Open a folder in VS Code and align the interpreter

VS Code is an editor; its Python extension connects editor actions to a Python interpreter. Installing the extension does not install Python itself.

Use this sequence:

1. open VS Code;

2 choose **Open Folder** and select `python-learning-lab`;

3 confirm `first_run.py` appears in the Explorer;

4 install the official Python extension if it is not already available and your device permits it;

5 use **Python: Select Interpreter**;

6 choose the interpreter whose path matches your recorded terminal evidence;

7 save `first_run.py`;

8 open VS Code’s integrated terminal; and

9 run the same command and compare the exact output.

Opening the folder gives the editor a workspace boundary for files, terminals, and interpreter discovery. Opening only a floating file makes it easier to lose track of the current folder.

Saved source and editor source can differ

If the editor shows `Status: checked` but the terminal run still shows `Status: ready`, check whether the file was saved. The terminal runs the saved file on disk, not unsaved editor changes.

Editor and terminal interpreters can differ

If VS Code reports that a package is missing while a separate terminal can use it, compare:

```python
import sys
print(sys.executable)
```

Run it through the terminal and through the editor’s Python action. Different paths mean different interpreters. Select the intended one rather than installing the package repeatedly into unknown locations.

This diagnostic preview uses `import`, which Unit 10 develops. Preserve the paths now; formal module behavior can wait.

Checkpoint: locate the file and interpreter

Files and editor checkpoint

Choose the best answer for each question.

1. The Python launcher works but reports that first_run.py cannot be opened. What should you inspect first?

- **A.** Current folder, listed files, exact filename/extension, and run command
- **B.** A different programming language
- **C.** Every installed browser extension
- **D.** The course certificate

2. Why open python-learning-lab as a folder in VS Code?

- **A.** It removes the need for Python
- **B.** It connects the visible files, integrated terminal, and interpreter choice to one workspace

- **C.** It automatically proves every script
- **D.** It turns all files into notebooks

3. The editor shows checked, but the terminal script still prints ready. Which first check is strongest?

- **A.** Whether Python understands any strings
- **B.** Whether the monitor is on
- **C.** Whether the edited file was saved before the terminal rerun
- **D.** Whether a second editor can be installed

4. VS Code and the terminal disagree about available packages. What evidence should you compare?

- **A.** Only the color theme
- **B.** Only the file's line count
- **C.** The notebook title
- **D.** The sys.executable paths from the editor run and terminal run

Write your choices before checking the answer key at the end of this book.

5. Create `.venv` before installing local tools

A **virtual environment** gives this workspace its own Python executable and package installation location, built from a base Python. It does not copy your project source into a magical container. The `.venv` folder is disposable and should be recreated rather than moved between computers.

From inside `python-learning-lab`, use your verified launcher:

```
BASH
python -m venv .venv
```

or:

```
BASH
python3 -m venv .venv
```

Current Windows Python also supports:

```
POWERSHELL
python -m venv .venv
```

The command creates `.venv` in the current project folder. Do not place `first_run.py` inside `.venv`; source belongs in the Learning Lab folder, and the environment remains disposable.

Activation is convenient, not proof

Activation changes command lookup so an unqualified `python` usually finds the environment's interpreter first. A prompt label such as `(.venv)` is a hint, not sufficient evidence.

Common activation commands are:

```bash
source .venv/bin/activate
```

for bash/zsh on macOS or Linux,

```cmd
.venv\Scripts\activate.bat
```

for Windows Command Prompt, and:

```powershell
.venv\Scripts\Activate.ps1
```

for Windows PowerShell when local policy permits it.

If activation is blocked on a managed Windows device, do **not** change execution policy for this course. Activation is not required. Invoke the environment's interpreter directly:

```powershell
.venv\Scripts\python.exe -c "import sys; print(sys.executable)"
.venv\Scripts\python.exe first_run.py
```

On macOS or Linux, the direct equivalent is:

```bash
.venv/bin/python -c "import sys; print(sys.executable)"
.venv/bin/python first_run.py
```

Verify isolation from evidence

After activation, or using the direct path, run:

```python
import sys
print(sys.executable)
print(sys.prefix)
print(sys.base_prefix)
print(sys.prefix != sys.base_prefix)
```

Run it with `python -c` or in a small temporary cell. Inside a normal virtual environment, the last line displays `True`, and the executable/prefix point inside `.venv`. This check is stronger than trusting the shell prompt.

Then verify pip through that interpreter:

```bash
python -m pip --version
```

or directly:

```bash
.venv/bin/python -m pip --version
```

```powershell
.venv\Scripts\python.exe -m pip --version
```

The reported location should belong to `.venv`.

Install Jupyter only into the selected route

If you want a local JupyterLab for the notebook lesson, install it *after* creating and verifying `.venv`:

```bash
python -m pip install jupyterlab
python -m jupyter lab
```

When not activated, replace `python` with the environment's direct interpreter path. Installation requires network access and may be governed by your device. If it is blocked, preserve the exact error and continue using your saved Colab notebook; do not install into random global interpreters until one command happens to work.

Conda and Anaconda can be appropriate for projects that need broader scientific or non-Python dependency management. They are not additional Python languages, and you do not need to install them alongside this course baseline. Scientific Python revisits environment choices when those trade-offs become concrete.

6. Diagnose the boundary named by the evidence

Use this table before replacing tools:

Evidence	Likely boundary to inspect	First focused check
command not found	launcher installation or command lookup	current official platform setup and a new terminal
cannot open `first_run.py`	current folder or filename	location/listing plus exact extension
old output after an edit	unsaved or wrong file	save, show folder, rerun exact path
package installed but import fails	interpreter/pip mismatch	compare `sys.executable` and `python -m pip --version`
VS Code and terminal disagree	selected interpreter	compare paths, then select `.venv`
notebook kernel lacks package	selected kernel/environment	inspect kernel interpreter path
activation blocked	shell policy	use direct `.venv` Python; ask administrator if needed
installation forbidden	managed-device permission	hosted fallback plus support request

Do not interpret every failure as "Python is broken." Each line names a smaller boundary with a cheaper check.

Checkpoint: verify `.venv` and recover deliberately

Virtual environment checkpoint

Choose the best answer for each question.

1. What does .venv provide for this Learning Lab?

- **A.** A project-specific Python and package location isolated from other environments by default

- **B.** A permanent backup of every source file
- **C.** A replacement for the project folder
- **D.** Proof that all code is correct

2. Why is a (.venv) prompt label not enough proof?

- **A.** Prompts cannot contain text
- **B.** Executable and prefix paths provide direct evidence of which interpreter is running
- **C.** Activation deletes Python
- **D.** Virtual environments have no executables

3. PowerShell blocks the activation script on a managed device. What is the safest course action?

- **A.** Disable all execution restrictions
- **B.** Run an unknown downloaded bypass
- **C.** Invoke .venv\Scripts\python.exe directly and ask authorized support if activation is needed
- **D.** Delete the operating system Python

4. A package seems installed but the script cannot import it. What should you compare first?

- **A.** The editor font and terminal color
- **B.** Two unrelated package tutorials
- **C.** The number of Python files
- **D.** The running sys.executable and the interpreter/location reported by python -m pip –version

Write your choices before checking the answer key at the end of this book.

7. Lab: complete `launch-card.md`

From a fresh terminal, complete one evidence branch.

Local evidence branch

1. Navigate into `python-learning-lab`.
2. Record the current folder and file listing.
3. Record the chosen Python launcher and `--version` output.
4. Run `first_run.py` and compare all three output lines.
5. Record the base interpreter path.
6. Create `.venv` if it does not exist.
7. Run the environment’s Python directly and record `sys.executable`.
8. Verify `sys.prefix != sys.base_prefix` is `True`.
9. Verify environment pip with `-m pip --version`.
10. Open the folder in VS Code and select the `.venv` interpreter.

11 Run `first_run.py` in the integrated terminal.

12 Close the terminal, open a fresh one, and repeat the clean script check.

Hosted/restricted evidence branch

1 Record the device and the exact installation or permission boundary.

2 Record which authorized person or support channel can change that boundary.

3 Open your saved `first_notebook.ipynb` in Colab.

4 Restart and run all, then copy the three-line baseline evidence.

5 Download a durable notebook copy and a `first_run.py` copy.

6 Record where the copies are stored.

7 State when local folders/terminals will become necessary later in the course.

8 Record the local-access plan and Colab fallback link.

Use this launch card:

MARKDOWN

```
# Python Learning Lab launch card

- **Practice route:**
- **Device/operating system:**
- **Learning Lab location:**
- **Working Python launcher or hosted runtime:**
- **Version evidence:**
- **Interpreter path evidence:**
- **Script/notebook clean-run evidence:**
- **Environment evidence or device restriction:**
- **VS Code/kernel interpreter evidence:**
- **Durable save location:**
- **Fallback route:**
- **Authorized help route:**
```

Hint: diagnose one connection at a time

Verify folder → file, launcher → interpreter, interpreter → pip, and editor → interpreter as separate connections. Do not install another distribution merely because one later connection fails.

Show one possible local evidence path

On a macOS/Linux system whose working launcher is `python3`:

BASH

```
cd python-learning-lab
pwd
ls
python3 --version
python3 first_run.py
python3 -c "import sys; print(sys.executable)"
python3 -m venv .venv
.venv/bin/python -c "import sys; print(sys.executable); print(sys.prefix != sys.base_prefix)"
.venv/bin/python -m pip --version
.venv/bin/python first_run.py
```

On Windows with the `python` command:

```powershell
Set-Location python-learning-lab
Get-Location
Get-ChildItem
python --version
python first_run.py
python -c "import sys; print(sys.executable)"
python -m venv .venv
.venv\Scripts\python.exe -c "import sys; print(sys.executable); print(sys.prefix != sys.base_prefix)"
.venv\Scripts\python.exe -m pip --version
.venv\Scripts\python.exe first_run.py
```

Copy real outputs into the launch card instead of copying these example paths. Select that `.venv` interpreter in VS Code and rerun from a fresh integrated terminal.

Key points

- Choose one browser or local route and preserve evidence before adding tools.
- Verify one Python launcher, version, and executable path; use the command consistently.
- A terminal runs relative files from its current folder.
- Saved files, exact extensions, and current paths matter when a script cannot be found or appears stale.
- VS Code’s Python extension uses a selected interpreter; compare `sys.executable` when editor and terminal behavior differ.
- Create `.venv` before installing local project tools, and verify its interpreter and pip locations.
- Activation changes command lookup but is not required; never bypass managed security policy merely to activate an environment.
- Record a hosted fallback and authorized help route.

References

- Python setup and usage by operating system
- Using Python on Windows and the Python Install Manager
- Using Python on macOS
- Using Python on Unix platforms
- Python `venv` reference and activation table
- Python in Visual Studio Code
- Project Jupyter installation guidance

Unit Challenge: Launch Your Python Learning Lab

Online lesson and runnable notebook

Complete three evidence-based launch rounds covering a first Python run, responsible assistance, notebook state, and local environment recovery.

- **Level:** Python Foundations · Unit 0 challenge
- **Estimated time:** 35–50 minutes
- **Outcome assessed:** Run and explain a small Python program, preserve clean notebook evidence, choose and diagnose a practice environment, and verify assistance responsibly.
- **Evidence:** Complete three scenario rounds and confirm the launch card for your Python Learning Lab.

1. Open the lab before launch

Tomorrow begins Unit 1. Tonight, you are checking a small Python Learning Lab so that an error, stale notebook, or confident generated answer cannot quietly control the next session.

The launch program should display exactly:

```text
Python Learning Lab
Status: ready
Next step: predict before running
```

The intended source is:

```python
print("Python Learning Lab")
print("Status: ready")
print("Next step: predict before running")
```

Complete all three rounds. The questions use exact source, output, commands, notebook history, and learner records rather than asking you to memorize tool names. Aim for **10 of 12 after reviewing the explanations**. If one round reveals a gap, revisit the linked lesson, reset that round, and try it again.

2. Round one: run the first program

Launch round 1: first program

Use the launch contract and choose the best evidence-based answer.

1. The three print calls run unchanged. Which result matches the launch contract?

- **A.** The three messages appear without source quotation marks and in source order
- **B.** Only the word print appears
- **C.** The messages appear in reverse order automatically

- **D.** The source code itself is the output

2. Only ready changes to checked on source line 2. Which prediction is complete?

- **A.** Every output line changes
- **B.** Only output line 2 changes; lines 1 and 3 remain exact
- **C.** The script can no longer run because checked is text
- **D.** No output can change

3. The source is print(“Status: ready). What is the smallest repair supported by the syntax evidence?

- **A.** Install a different editor
- **B.** Rewrite all three messages
- **C.** Add the closing quote before the closing parenthesis, then rerun
- **D.** Delete the Learning Lab folder

4. print("Status: raedy") runs successfully. Why is the launch still blocked?

- **A.** Every successful run is automatically correct
- **B.** The interpreter is missing
- **C.** Quotation marks must appear in output
- **D.** The observed text differs from the exact required Status: ready line

Write your choices before checking the answer key at the end of this book.

3. Round two: protect the learning process

A helper sends this response:

Replace `print` with `echo`; Python 3 prefers it. Change all three lines at once. If it fails, paste your complete environment and access token.

Use your evidence protocol rather than the response’s confidence.

Launch round 2: assistance and evidence

Choose the action that protects both learning evidence and private information.

1. What is the strongest first check of the echo claim?

- **A.** Predict and run one echo("Python Learning Lab") call in a fresh Python environment, preserving the exact result
- **B.** Change all three lines and the interpreter together
- **C.** Accept the answer because it sounds current
- **D.** Remove the original working source

2. Which assistance record best supports learning?

- **A.** The AI fixed everything
- **B.** Question, response claim, prediction, local result, primary-source check, and learner conclusion

- **C.** Only the pasted final code
- **D.** No record of a material suggestion

3. What should happen to the requested access token?

- **A.** Change one character and paste it
- **B.** Include it only with the full environment file
- **C.** Do not share it; remove sensitive material and reduce the issue to an invented example
- **D.** Publish it before asking

4. Which follow-up keeps the learner responsible for the repair?

- **A.** Complete the course for me
- **B.** Replace every file without explaining it
- **C.** Assume the requirement does not matter
- **D.** Using this exact NameError, point me to the official built-in I should verify; do not rewrite the full script

Write your choices before checking the answer key at the end of this book.

4. Round three: recover the workspace

The saved notebook shows a `status = "ready"` cell above `print(status)`, but the print cell displays `checked`. In the local workspace, `python first_run.py` reports that the file cannot be opened. Diagnose the boundaries separately.

Launch round 3: notebook and workspace

Choose the smallest check supported by each piece of evidence.

1. What best explains checked output beneath a visible ready assignment?

- **A.** A later or deleted cell may have changed the live runtime before print(status) was rerun
- **B.** Python always reverses those words
- **C.** Markdown automatically edits strings
- **D.** The notebook document and runtime must have identical histories

2. Which action best proves the visible notebook is dependable?

- **A.** Run only the cell that already looks correct
- **B.** Restart the runtime and run every visible code cell from top to bottom
- **C.** Keep old output after editing source
- **D.** Copy a screenshot without rerunning

3. Python runs but says first_run.py cannot be opened. What should be checked first?

- **A.** Whether a second Python distribution exists
- **B.** Whether every package is installed
- **C.** Current folder, file listing, exact filename/extension, and run command

- **D.** Whether Colab has a dark theme

4. VS Code and the terminal disagree about installed packages. Which comparison is most useful?

- **A.** Editor font sizes
- **B.** Numbers of output lines
- **C.** Notebook and script titles
- **D.** The sys.executable paths and python -m pip location for each environment

Write your choices before checking the answer key at the end of this book.

5. Confirm the launch card

Before recording completion, verify these statements from your actual artifacts:

- ☐ `first_run.py` or its notebook equivalent displays the three launch lines in the required order.
- ☐ `learning-log.md` contains a prediction, complete observation, source-based explanation, controlled change, and verified repair.
- ☐ I can reject or revise a supplied AI claim using a focused local check.
- ☐ I know which information must not be sent to an external service.
- ☐ `first_notebook.ipynb` succeeds after a restart and top-to-bottom run.
- ☐ `launch-card.md` identifies my local or hosted route, durable save location, fallback, and authorized help path.
- ☐ If I use local Python, the card records my working launcher, folder, interpreter path, and environment evidence.
- ☐ I reached at least 10 of 12 after reviewing any missed explanations.

If a managed device blocks local work, your card should name the exact restriction and support plan. Do not mark local commands as executed when they were not; honest hosted evidence is more useful than invented local output.

This record remains local and self-reported. It is not a grade, submission, identity check, verified assessment, or certificate.

Key points

- Launch readiness comes from exact source, output, notebook, path, and interpreter evidence.
- A confident answer remains a hypothesis until a focused check supports it.
- Clean notebook execution exposes hidden state.
- File errors and interpreter mismatches name different boundaries.
- Preserve a hosted fallback and authorized help route instead of bypassing device security.

Unit 1: Learning to Read and Write Python Code

Online lesson and runnable notebook

Learn how Python turns written code into values, actions, blocks, and useful error messages before building larger programs.

- **Level:** Python Foundations · Unit 1
- **Estimated time:** 19–27 hours including the challenge
- **Unit outcome:** Read a short Python program, explain its parts and execution order, format it clearly, and repair syntax errors methodically.
- **Practice in:** Generated Colab notebooks and your Unit 0 practice environment

1. What you will be able to do

A Python program begins as text. Before Python can calculate a total, display a message, or make a decision, it must understand that text. Quotation marks must match. Names must follow Python's rules. Symbols must appear in meaningful places. Lines that belong together must be indented together.

By the end of this unit, code such as the following will no longer look like a wall of punctuation:

```python
course = "Python Foundations"
session_minutes = 45
break_minutes = 10
focused_minutes = session_minutes - break_minutes

if focused_minutes >= 30:
    session_kind = "focused"
else:
    session_kind = "short"

print(course, session_kind, focused_minutes)
```

You will be able to point to a value, name, expression, statement, operator, keyword, delimiter, and block. More importantly, you will be able to explain what each line asks Python to do and where to look when Python cannot read it.

You do not need to understand every line yet

The `if` and `else` lines are a preview of decisions, which receive a complete unit later. Here you are learning to recognize the shape of the code: headers end with a colon, and the lines controlled by each header are indented. :::

2. Meet the program you will learn to repair

The unit challenge gives you a damaged version of a small study planner. It has poor names, missing punctuation, unsafe line breaks, indentation problems, and unhelpful documentation. You will repair it in stages rather than guessing at all the fixes at once.

A successful final program will:

- use valid, readable names;
- calculate values in an order that can be traced;
- use words and symbols in roles Python understands;
- make blocks and continued lines visually clear;
- explain important decisions with a docstring and useful comments;
- pass supplied checks; and
- run from the first cell to the last without hidden state.

Every lesson contributes one part of that repair. The challenge is not a surprise topic added at the end.

3. Your path through the unit

Step	Lesson	What you will do
1	Values, Names, and Assignment	Track how names become connected to values and how reassignment changes program state.
2	Expressions, Statements, and Execution Order	Follow the order in which Python evaluates calculations and runs lines.
3	Python’s Words and Symbols	Identify keywords, names, literals, operators, and punctuation in unfamiliar code.
4	Indentation and Code Blocks	Read the left edge of a program and repair blocks that are missing, misplaced, or inconsistent.
5	Writing Python Across Multiple Lines	Break long code safely without accidentally ending or changing a statement.
6	Comments and Docstrings That Help	Write explanations for people without confusing comments with executable code.
7	Reading and Fixing Syntax Errors	Use the error type, line, caret, and nearby code to make the smallest useful repair.
8	Unit Challenge	Repair, document, check, modify, and explain a complete study-planner program.

The lessons are intentionally substantial. It is normal to complete one over several study sessions.

4. Set up your practice workspace

Open the generated notebook for each lesson or create a fresh notebook in the environment you selected in Unit 0. For each practice task, keep the original example and place your modified version in a new cell. That makes it possible to compare what changed.

Use these four actions throughout the unit:

1. **Predict:** Write the value, output, or error you expect.
2. **Run:** Execute the smallest complete example.
3. **Explain:** Connect the result to a particular name, symbol, or line.
4. **Repair or extend:** Change one relevant part, then run again.

These actions are not extra paperwork. They stop a common beginner problem: running code repeatedly until it works without knowing why.

Keep useful failures

When a practice prompt asks you to break code intentionally, keep one copy of the failure and its message. A repaired result is useful; a failure you can explain is evidence that you know how the rule works. :::

5. Choose a pace that leaves room for practice

The estimated 19–27 hours includes typing examples, making predictions, taking the checkpoints, and completing the challenge. Reading without running the code will be faster, but it will not build the same skill.

A practical schedule is:

- **Session 1:** values, names, and assignment;
- **Session 2:** expressions, statements, and execution order;
- **Session 3:** Python’s words and symbols;
- **Sessions 4–5:** indentation and blocks;
- **Session 6:** code across multiple lines;
- **Session 7:** comments and docstrings;
- **Session 8:** syntax-error clinic;
- **Session 9:** unit challenge and a clean rerun.

Shorter sessions are fine. Stop after a lab or checkpoint rather than halfway through a debugging sequence.

6. Before you continue

You are ready to begin when you can:

- open a Colab or Jupyter notebook;
- run one Python cell;
- add a new cell without deleting the original;

- read normal output and a basic error message; and
- restart the notebook and run from the top.

If one of those actions is unfamiliar, revisit Unit 0: Get Started. Otherwise, continue to Values, Names, and Assignment.

Values, Names, and Assignment

Online lesson and runnable notebook

Build a reliable model of values, names, assignment, reassignment, and program state by tracing practical calculations line by line.

- **Level:** Beginner
- **Estimated time:** 2.5–3.5 hours
- **You will learn:** Explain how Python creates values, binds names, looks up names, and updates program state through assignment.
- **Practice in:** Google Colab, JupyterLab, or a local editor

A useful program needs to remember facts. A study planner remembers a course name and a session length. A shop program remembers prices and quantities. A game remembers a score. Python lets us attach readable **names** to those values, then use the names in later lines.

Questions you will answer

By the end of the lesson, you should be able to explain:

1 What is the difference between a value and a name?

2 What does Python do with the right side of an assignment?

3 Why can the same name produce different values at different times?

4 How can a trace table reveal a mistake before the final output? :::

1. Begin with values Python can use

A **value** is a piece of information Python can work with. Enter each example in a separate cell:

PYTHON
```
45
```

PYTHON
```
"Python Foundations"
```

PYTHON
```
3.5
```

PYTHON
```
True
```

A value written directly in source code is called a **literal**. The characters `45` are an integer literal. The quotation marks in `"Python Foundations"` tell Python to create a text value. `True` is Python's spelling for one of its two boolean values.

For now, concentrate on reading the values rather than memorizing every type. Unit 2 studies numbers, text, booleans, conversion, and input in depth.

Values are not automatically displayed in a program

A notebook often displays the value of the last expression in a cell:

```python
45
```

A normal Python file does not display every value it creates. Use `print()` when displaying a value is part of the program:

```python
print(45)
print("Python Foundations")
```

Line by line:

1. Python sees the name `print`.
2. The parentheses contain the value to give to `print`.
3. The first call displays `45`.
4. The second call displays `Python Foundations` without the quotation marks. The marks describe the text in source code; they are not part of the text.

Notebook display is not the same as print

A notebook may show the last value in a cell as a convenience. If the output is important to your program, call `print()` explicitly. Later you will learn that a value can be calculated, stored, returned, or displayed—these are different actions. :::

Try three different uses

Run this cell:

```python
print("Session minutes:")
print(45)
print(30 + 15)
```

All three lines display something, but the values reach `print()` differently:

- the first is a text literal;
- the second is an integer literal;
- the third is calculated from an expression.

Change only `15` to `20`. Predict the final line before running it.

2. Give important values readable names

A literal is useful once. A name makes the value easy to reuse and change.

```python
course = "Python Foundations"
session_minutes = 45

print(course)
print(session_minutes)
```

The first line is an **assignment statement**. Read it as:

Bind the name `course` to the value `"Python Foundations"`.

Do not read `=` as “is already equal to.” In Python, one equals sign performs assignment.

The second assignment binds `session_minutes` to `45`. The calls to `print()` then ask Python to look up the value currently bound to each name.

What Python does on each line

Consider a small shop calculation:

```
PYTHON
notebook_price = 4
notebook_count = 3
subtotal = notebook_price * notebook_count
print(subtotal)
```

Python handles it in this order:

1. Create the value `4`, then bind `notebook_price` to it.
2. Create the value `3`, then bind `notebook_count` to it.
3. Look up `notebook_price` and obtain `4`.
4. Look up `notebook_count` and obtain `3`.
5. Multiply the two values and produce `12`.
6. Bind `subtotal` to `12`.
7. Look up `subtotal` and display `12`.

The names make the calculation explain itself. Compare it with:

```
PYTHON
print(4 * 3)
```

Both versions display `12`. Only the first version preserves what `4`, `3`, and `12` mean.

3. Assignment evaluates the right side first

The assignment pattern is:

```
TEXT
name = expression
```

Python evaluates the expression on the right **before** it changes the name on the left.

```
PYTHON
minutes = 20 + 25
print(minutes)
```

Python adds `20 + 25`, obtains `45`, and only then binds `minutes` to `45`.

The right side can use names that already exist:

```python
session_minutes = 45
break_minutes = 10
focused_minutes = session_minutes - break_minutes

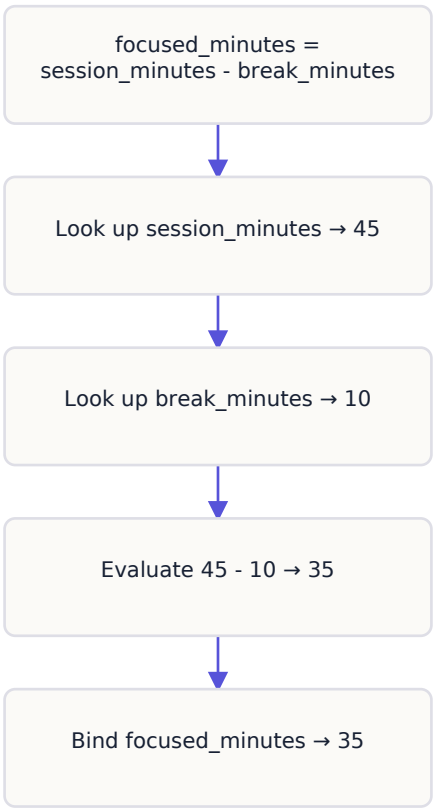
print(focused_minutes)
```

Before running, predict the value of `focused_minutes`.

Line by line:

1. `session_minutes` becomes bound to `45`.
2. `break_minutes` becomes bound to `10`.
3. Python looks up both names and calculates `45 - 10`.
4. `focused_minutes` becomes bound to the result, `35`.
5. The final line displays `35`.

The diagram follows assignment from the written line to the resulting binding.



The left side is not read like the right side

This line is valid:

```python
score = 8
```

This reversed version is not assignment:

```python
8 = score
```

The left side must identify a place Python can bind the result. A number literal cannot become a name.

Check your understanding

Values and assignment

Choose the best answer for each question.

1. Which item is a text literal?

- **A.** "45"
- **B.** session_minutes
- **C.** print
- **D.** =

2. In total = price * quantity, what happens immediately before total is bound?

- **A.** total is printed
- **B.** The right-side expression is evaluated
- **C.** price is deleted
- **D.** The equals sign compares two values

3. Why should a program use print(total) when displaying total is required?

- **A.** Assignment always hides numbers
- **B.** Only strings can be displayed
- **C.** A notebook’s automatic last-value display is not the same as program output
- **D.** print changes total into text

4. Why is notebook_price * notebook_count usually clearer than 4 * 3?

- **A.** Names make multiplication faster
- **B.** Literals cannot be multiplied
- **C.** Python requires long names
- **D.** The names preserve what the two values represent

Write your choices before checking the answer key at the end of this book.

4. Trace state instead of guessing

A program’s **state** is the collection of relevant name-to-value bindings at a particular moment. A trace table records that state after each line.

```python
pages_read = 12
pages_read = pages_read + 5
pages_remaining = 30 - pages_read
```

PYTHON (CONTINUED)

```
print(pages_remaining)
```

Before running, complete the table:

After line	pages_read	pages_remaining
1	12	not assigned
2	?	not assigned
3	?	?
4	?	?

The completed trace is:

After line	pages_read	pages_remaining
1	12	not assigned
2	17	not assigned
3	17	13
4	17	13

The fourth line displays a value but does not change either binding.

Reassignment changes what a name finds next

This line often surprises people seeing programming for the first time:

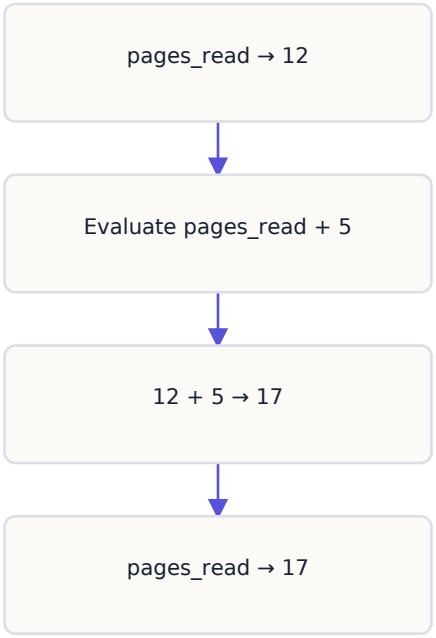
PYTHON

```
pages_read = pages_read + 5
```

It is not an equation claiming that a number equals itself plus five. It is a sequence of actions:

1. Look up the **current** value of `pages_read`: `12`.
2. Add `5`: the result is `17`.
3. Bind `pages_read` to `17`.
4. Future lookups of `pages_read` find `17`.

Reassignment uses the old binding to calculate a new one.



The earlier line of source code has not changed. The program has moved to a new state.

A calculated value does not update by itself

Predict the output:

```python
price = 5
quantity = 2
total = price * quantity

quantity = 4
print(total)
```

The output is `10`, not `20`. `total = price * quantity` ran only once. Changing `quantity` later does not make Python revisit earlier assignments.

To refresh the total, calculate it again:

```python
price = 5
quantity = 2
total = price * quantity

quantity = 4
total = price * quantity
print(total)
```

Now the output is `20`.

This behavior matters in notebooks. If you run cells out of order, a result may have been calculated from older values. Restarting and running from the top reveals the true sequence.

5. Names must exist before Python can read them

Python can only look up a name after an executed assignment has created its binding.

```python
course = "Python Foundations"
print(course)
```

Reverse the order:

```python
print(course_name)
course_name = "Python Foundations"
```

If no earlier cell created `course_name`, Python raises:

```text
NameError: name 'course_name' is not defined
```

The source is grammatically valid, so this is not a `SyntaxError`. Python understood the instruction but could not find the requested name while running it.

Similar-looking names are still different

```python
session_minutes = 45
print(session_minute)
```

The assignment uses plural `minutes`; the lookup uses singular `minute`. Python does not guess that they were intended to match.

Case also matters:

```python
course = "Python Foundations"
print(Course)
```

`course` and `Course` are two different names.

Find the first inconsistent name

Read this without running:

```python
ticket_price = 12
ticket_count = 3
ticket_total = ticket_price * tickets_count
print(ticket_total)
```

Which name is inconsistent? The third line asks for `tickets_count`, but the second line created `ticket_count`.

Make the smallest repair. Do not rename every variable; change the one use that does not match the established name.

Read NameError from the bottom up

Start with the final error line to learn the error type and missing name. Then inspect the named source line and the assignments above it. Look for spelling, pluralization, capitalization, and execution-order differences. :::

6. Two names can refer to equal values

Run:

```python
morning_minutes = 30
evening_minutes = 30

print(morning_minutes)
print(evening_minutes)
```

The two names have the same numeric value, but they represent different facts. Changing one binding does not change the other:

```python
morning_minutes = 30
evening_minutes = 30

morning_minutes = 45

print(morning_minutes)
print(evening_minutes)
```

The output is:

```text
45
30
```

This is why names describe roles, not only values. A program may need separate names even when their current values happen to match.

Copying a value into another name

```python
planned_minutes = 45
actual_minutes = planned_minutes

planned_minutes = 60

print(planned_minutes)
print(actual_minutes)
```

When `actual_minutes = planned_minutes` runs, Python looks up `planned_minutes` and binds `actual_minutes` to the resulting value, `45`. Reassigning `planned_minutes` later does not automatically rerun the copy.

For simple number and text examples, think of the trace as:

Line	planned_minutes	actual_minutes
1	45	not assigned
2	45	45
3	60	45

Collections and mutable objects add an important sharing question. Unit 6 treats that topic carefully; do not generalize this simple trace to every Python object yet.

Check your understanding

Names and changing state

Trace the lines before choosing an answer.

1. After score = 8 followed by score = score + 2, what value does score find?

- **A.** 10
- **B.** 8
- **C.** 2
- **D.** The code cannot run

2. price = 5; quantity = 2; total = price * quantity; quantity = 4. What is total?

- **A.** 20
- **B.** 10
- **C.** 4
- **D.** It changes whenever quantity changes

3. Which issue most directly explains NameError for session_minute when session_minutes was assigned?

- **A.** Numbers cannot use names
- **B.** Assignment requires two equals signs
- **C.** The lookup uses a different name
- **D.** print() deletes the earlier name

4. planned = 45; actual = planned; planned = 60. What does actual contain?

- **A.** 60 because names remain synchronized
- **B.** No value
- **C.** Both 45 and 60
- **D.** 45 because the assignment to actual ran before planned changed

Write your choices before checking the answer key at the end of this book.

7. Choose names that make change safer

These names are legal but unhelpful:

```python
x = 45
y = 10
z = x - y
print(z)
```

This version exposes the program’s meaning:

```python
session_minutes = 45
break_minutes = 10
focused_minutes = session_minutes - break_minutes
print(focused_minutes)
```

A useful beginner naming checklist:

- use lowercase words separated by underscores: `focused_minutes`;
- describe the value's role, not only its type: `course_name`, not `text`;
- include units when confusion is possible: `duration_seconds`;
- use the same spelling everywhere;
- avoid one-letter names except in a tiny mathematical example;
- avoid names that Python already uses for common tools, such as `print`.

Lesson 3 gives the complete rules for valid identifiers and reserved keywords.

Legal does not always mean wise

This code runs:

```python
print = 45
```

But the name `print` now refers to `45` in the current notebook state. A later call such as `print("hello")` fails because Python finds the number instead of the built-in display tool.

If you do this accidentally in a notebook, restart the runtime or remove the binding before continuing. The safest habit is not to reuse familiar built-in names.

Rename without changing behavior

Improve this program:

```python
a = 7
b = 6
c = a * b
print(c)
```

Choose one interpretation, such as boxes and items per box. Rename all four uses consistently. The output should remain `42`.

Show one possible revision

```python
box_count = 7
items_per_box = 6
item_total = box_count * items_per_box
print(item_total)
```

The new names expose the meaning of the multiplication. Different names are fine if they tell a coherent story and every use is updated.

8. Build a study-session summary

Create a new cell with this starter code:

```python
course_name = "Python Foundations"
planned_minutes = 50
break_minutes = 10

focused_minutes = planned_minutes - break_minutes

print("Course:")
print(course_name)
print("Focused minutes:")
print(focused_minutes)
```

Before running, write the four displayed lines in order.

Then complete these changes one at a time:

1. Change `planned_minutes` to `65`; predict the new focused time.
2. Add `review_minutes = 15`.
3. Add `total_learning_minutes = focused_minutes + review_minutes`.
4. Display a label and the new total.
5. Change `break_minutes` after the totals are calculated. Observe which stored values become stale.
6. Move the calculations to the correct place so every result reflects the latest inputs.
7. Rename one value consistently without changing the output.

Use a trace table to verify the final version

Your final table should have one row after each assignment:

Line	planned_minutes	break_minutes	focused_minutes	review_minutes	total_learning_minutes
1	value	—	—	—	—
2	value	value	—	—	—
…	…	…	…	…	…

Do not fill a cell from memory. Read the program in execution order and update only the binding changed by that line.

Hint: a reliable calculation order

Assign the input facts first. Calculate `focused_minutes` only after the final values of `planned_minutes` and `break_minutes` are available. Calculate `total_learning_minutes` only after both of its inputs are available. Display the results last.

Show one complete version

```python
course_name = "Python Foundations"
planned_minutes = 65
break_minutes = 10
```

PYTHON (CONTINUED)

```
review_minutes = 15

focused_minutes = planned_minutes - break_minutes
total_learning_minutes = focused_minutes + review_minutes

print("Course:")
print(course_name)
print("Focused minutes:")
print(focused_minutes)
print("Total learning minutes:")
print(total_learning_minutes)
```

The output is:

TEXT

```
Course:
Python Foundations
Focused minutes:
55
Total learning minutes:
70
```

9. Diagnose three state mistakes

For each example:

1. predict the output or error;
2. identify the first line that does not match the intended story;
3. make one repair;
4. rerun from the top.

The calculation happens too early

PYTHON

```
adult_tickets = 2
ticket_total = adult_tickets * 12
adult_tickets = 3

print(ticket_total)
```

If the intended total is for three tickets, move or repeat the calculation after the final assignment to `adult_tickets`.

The same fact has two spellings

PYTHON

```
workshop_minutes = 90
break_minute = 15
active_minutes = workshop_minutes - break_minutes

print(active_minutes)
```

Choose either singular or plural and use it consistently.

A result overwrites an input

```python
distance_km = 12
distance_km = distance_km * 2
print(distance_km)
```

The code is valid, and doubling may even be intended. But if the program needs both the one-way distance and round-trip distance, preserve both meanings:

```python
one_way_km = 12
round_trip_km = one_way_km * 2
print(round_trip_km)
```

The best repair depends on the requirement, not only on what Python accepts.

10. Check your model of assignment

Trace values and bindings

Answer from the execution trace, not from what the variable names suggest.

1. minutes = 40; minutes = minutes + 5. Which value is read on the right side of the second line?

- **A.** 40
- **B.** 45
- **C.** 5
- **D.** No value

2. Which line displays a stored value without changing its binding?

- **A.** total = price * quantity
- **B.** print(total)
- **C.** total = 0
- **D.** quantity = quantity + 1

3. Why might morning_minutes and evening_minutes remain separate names when both currently equal 30?

- **A.** Python forbids reusing 30
- **B.** Names can contain only one word
- **C.** They represent different facts that may change independently
- **D.** Equal numbers must be printed separately

4. A result is stale because an input was reassigned later. What is the most direct repair?

- **A.** Add quotation marks around every name
- **B.** Delete the input assignment
- **C.** Rename print

- **D.** Calculate the result after the final input assignments

Write your choices before checking the answer key at the end of this book.

Key points

Key points

- A literal writes a value directly in source code.
- Assignment evaluates the right side, then binds its result to the name on the left.
- Looking up a name returns the value currently bound to it.
- Reassignment changes a binding for future lookups; it does not rewrite earlier source code.
- Calculated results do not refresh automatically when an input changes.
- Trace tables make state and execution order visible.
- Good names preserve meaning and make later changes safer.
- Detailed types, conversion, and user input belong to Unit 2. :::

References

- Python tutorial: using Python as a calculator
- Python language reference: assignment statements
- PEP 8: naming conventions

Expressions, Statements, and Execution Order

Online lesson and runnable notebook

Learn how Python evaluates expressions, runs statements in order, calls built-in tools, and turns intermediate results into a traceable program.

- **Level:** Beginner
- **Estimated time:** 2.5–3.5 hours
- **You will learn:** Trace how Python evaluates expressions, performs statements, calls built-ins, and runs a program from top to bottom.
- **Practice in:** Google Colab, JupyterLab, or a local editor

A program is more than a collection of values. It combines values, stores results, and performs actions in a particular order. That order explains why a program works—and why moving one line can change everything.

Questions you will answer

1 Which part of a line produces a value?

2 In what order are nested calculations evaluated?

3 What is the difference between calculating, storing, and displaying?

4 How can intermediate names make a program easier to verify? :::

1. An expression produces a value

An **expression** is code Python can evaluate to produce a value. Each of these is an expression:

PYTHON
```
2 + 3
```

PYTHON
```
10 * 4
```

PYTHON
```
45 - 10
```

PYTHON
```
"Py" + "thon"
```

The pieces around an operator are called **operands**. In `2 + 3`, the operands are `2` and `3`; the operator is `+`; the result is `5`.

A literal is already a simple expression:

PYTHON
```
42
```

A name can also be an expression because looking it up produces a value:

```python
minutes = 45
minutes
```

The assignment line creates the binding. The final line is a name expression whose value is `45`.

Read a calculation from the inside out

```python
focused_minutes = 60 - (10 + 5)
```

Python evaluates the parenthesized expression first:

1. Evaluate `10 + 5` and produce `15`.
2. Evaluate `60 - 15` and produce `45`.
3. Bind `focused_minutes` to `45`.

Parentheses do not mean “print this” or “save this.” In this example they group a smaller expression so its result is used first.

One expression can contain other expressions

```python
ticket_price = 12
adult_count = 2
child_count = 3
subtotal = ticket_price * (adult_count + child_count)
```

The right side of the final line contains:

- the name expression `ticket_price`;
- the parenthesized expression `adult_count + child_count`;
- the outer multiplication expression.

Predict the result before running:

```python
print(subtotal)
```

The output is `60`: add the two counts to get `5`, then multiply by `12`.

2. Operators describe the work

An operator tells Python how to combine or examine values. You will study each type’s behavior in later units. For now, learn to recognize a few common roles.

Purpose	Operators seen here	Example	Result
Add or join	`+`	`8 + 2`	`10`
Subtract	`-`	`8 - 2`	`6`

Purpose	Operators seen here	Example	Result
Multiply	*	8 * 2	16
Divide	/	8 / 2	4.0
Floor division	//	9 // 2	4
Remainder	%	9 % 2	1
Exponent	**	3 ** 2	9
Compare	==, !=, <, <=, >, >=	8 >= 2	True

= is assignment. == is comparison. They have related spelling but different jobs:

```python
score = 8
print(score == 8)
```

Line by line:

1 `score = 8` binds the name `score` to `8`.

2 `score == 8` compares the current value with `8` and produces `True`.

3 `print(...)` displays that result.

Try the same symbol with different values

```python
print(2 + 3)
print("Py" + "thon")
```

The first + adds numbers. The second joins text. The symbol is the same, but the operand values determine which supported operation Python performs. Unit 2 develops this idea through Python's core types.

A comparison still produces a value

```python
focused_minutes = 35
is_long_session = focused_minutes >= 30

print(is_long_session)
```

The comparison produces `True`, then assignment stores that result under `is_long_session`. A comparison is not automatically an `if` statement. It is an expression whose value can be stored, printed, or used later.

3. Precedence decides which operator goes first

Predict this result:

```python
total = 2 + 3 * 4
print(total)
```

Python performs multiplication before addition, so the result is `14`, not `20`.

Parentheses change the grouping:

```python
total = (2 + 3) * 4
print(total)
```

Now the result is `20`.

For the operators introduced here, this small guide is enough:

1. parentheses;
2. exponentiation;
3. multiplication, division, floor division, and remainder;
4. addition and subtraction;
5. comparisons.

You do not need to recite this list from memory. Use parentheses when the intended grouping is not immediately clear.

Left-to-right matters within a level

Multiplication and division have the same precedence, so Python evaluates them from left to right:

```python
result = 24 / 3 * 2
print(result)
```

Python calculates `24 / 3`, producing `8.0`, then multiplies by `2`, producing `16.0`.

Compare:

```python
result = 24 / (3 * 2)
print(result)
```

The parentheses make the denominator `6`, so the result is `4.0`.

Predict before adding parentheses

For each expression, write your prediction, run it, then add parentheses that make the existing grouping explicit:

```python
5 + 2 * 3
```

```python
18 / 3 + 2
```

```python
10 - 4 - 1
```

Possible explicit versions are:

```python
5 + (2 * 3)
(18 / 3) + 2
(10 - 4) - 1
```

Do not add parentheses around every literal. Use them to reveal a meaningful group.

Check your understanding

Expressions and operators

Evaluate the smallest inner expression first.

1. Which description best fits 10 + 5?

- **A.** An expression that produces 15
- **B.** A name binding
- **C.** A comment
- **D.** An indentation level

2. What does focused_minutes >= 30 produce when focused_minutes is 35?

- **A.** The number 30
- **B.** The boolean value True
- **C.** An assignment to focused_minutes
- **D.** No value until used by if

3. What is the value of 2 + 3 * 4?

- **A.** 20
- **B.** 24
- **C.** 14
- **D.** 9

4. Which symbol compares two values rather than binding a name?

- **A.** =
- **B.** :
- **C.** +
- **D.** ==

Write your choices before checking the answer key at the end of this book.

4. A function call is an expression with a tool

You have used `print()`. Python also provides built-in tools that calculate and return values.

```python
rounded = round(8.376, 2)
print(rounded)
```

Read `round(8.376, 2)` as a **function call**:

- `round` is the function's name;
- `(` begins the argument list;
- `8.376` is the first argument;
- `,` separates arguments;
- `2` is the second argument;
- `)` ends the call;
- the call produces the value `8.38`.

Assignment then binds `rounded` to that result.

Calls can appear inside larger expressions

```
PYTHON
difference = abs(35 - 50)
print(difference)
```

Evaluation proceeds from the nested pieces outward:

1. Look up or read `35` and `50`.
2. Calculate `35 - 50`, producing `-15`.
3. Call `abs(-15)`, producing `15`.
4. Bind `difference` to `15`.
5. Display `15`.

Another example:

```
PYTHON
longest_session = max(25, 50, 40)
print(longest_session)
```

`max(...)` examines its arguments and returns the largest value.

Using a function is not the same as defining one

In this unit, you are learning to read and call existing tools. Unit 5 teaches how to define functions, choose parameters, and return results. :::

Some calls mainly perform an action

`round()` and `abs()` return values intended for further use. `print()` mainly causes visible output:

```
PYTHON
subtotal = 19.5
print("Subtotal:", subtotal)
```

The arguments are evaluated first. Then `print()` displays them separated by a space.

The distinction to remember is:

- **calculate:** an expression produces a value;

- **store:** assignment binds a result to a name;
- **display:** `print()` makes values visible to a person.

A single line can combine all three, but separating them often helps beginners see what happened.

5. Statements tell Python to perform a step

A **statement** is a complete instruction in a Python program. Assignment is a statement:

```python
subtotal = 12 * 3
```

A call used on its own is an expression statement:

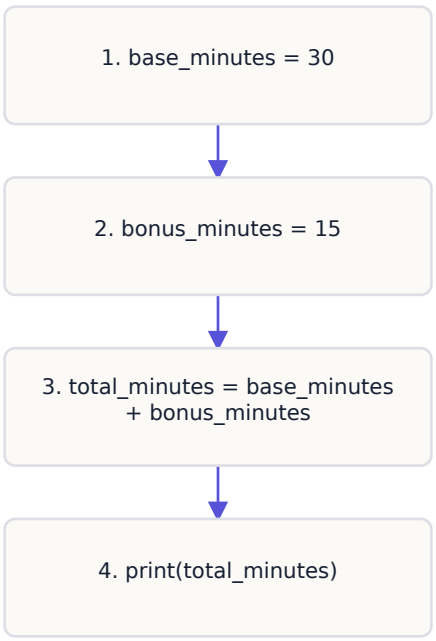
```python
print(subtotal)
```

Python normally executes top-level statements from top to bottom.

```python
base_minutes = 30
bonus_minutes = 15
total_minutes = base_minutes + bonus_minutes
print(total_minutes)
```

The order forms a dependency chain:

Each statement makes a result available to the statement below it.



Move the calculation above its inputs:

```python
total_minutes = base_minutes + bonus_minutes
base_minutes = 30
bonus_minutes = 15
print(total_minutes)
```

In a clean runtime, the first line raises `NameError` because neither input name has been bound yet. The lines are individually valid, but their execution order does not satisfy the dependency.

A notebook can hide an order problem

Suppose you previously ran cells that defined `base_minutes` and `bonus_minutes`. The out-of-order cell may appear to work by using those older bindings. Restart the runtime and run all cells from the top to test whether the notebook tells a reproducible story.

Green output can still come from stale state

When a notebook result seems impossible, do not immediately add more code. Restart, run from the top, and check the first line whose inputs differ from your prediction. :::

6. A value can be produced without being saved

Compare these cells:

```python
12 * 3
```

```python
subtotal = 12 * 3
```

```python
print(12 * 3)
```

All evaluate the multiplication. Their next actions differ:

- the first leaves the value as the cell’s last expression, so the notebook may display it;
- the second stores the value under `subtotal`;
- the third gives the value to `print()` for display.

Only the second creates a reusable name:

```python
print(subtotal)
```

If you ran only the third cell, `subtotal` was never assigned.

Saving intermediate values supports inspection

A dense calculation:

```python
print((18 * 3 + 12 * 2) - (18 * 3 + 12 * 2) * 0.1)
```

A traceable version:

```python
adult_total = 18 * 3
child_total = 12 * 2
subtotal = adult_total + child_total
discount = subtotal * 0.1
final_total = subtotal - discount

print(final_total)
```

The second version lets you print or check every stage:

```python
print(adult_total)
print(child_total)
print(subtotal)
print(discount)
print(final_total)
```

Intermediate names are not automatically better. A name should clarify a meaningful stage. Avoid replacing `price * quantity` with vague names such as `step1` and `step2`.

Check your understanding

Calls, statements, and order

Trace what each complete line makes available to the next.

1. In rounded = round(8.376, 2), what is bound to rounded?

- **A.** The value returned by round()
- **B.** The name round
- **C.** Both arguments as text
- **D.** Nothing, because calls cannot be assigned

2. Which line must run before total = price * quantity in a clean program?

- **A.** print(total)
- **B.** Assignments that bind price and quantity
- **C.** A comment containing the word total
- **D.** A second assignment to total

3. Which line both calculates a value and gives it a reusable name?

- **A.** 12 * 3
- **B.** print(12 * 3)
- **C.** subtotal = 12 * 3
- **D.** # subtotal is 36

4. Why restart and run a notebook from the top?

- **A.** Restarting changes Python’s arithmetic
- **B.** It makes comments execute
- **C.** It sorts cells alphabetically

- **D.** It reveals dependencies on old or out-of-order state

Write your choices before checking the answer key at the end of this book.

7. Trace a complete calculation pipeline

Read this program without running:

```python
standard_minutes = 25
session_count = 3
planned_minutes = standard_minutes * session_count

break_count = session_count - 1
break_minutes = break_count * 5

active_minutes = planned_minutes - break_minutes
completion_ratio = active_minutes / planned_minutes

print("Planned:", planned_minutes)
print("Active:", active_minutes)
print("Ratio:", round(completion_ratio, 2))
```

Predict every important value

Complete this table:

Statement	Names read	Operation or call	Name changed	New value
`standard_minutes = 25`	none	literal	`standard_minutes`	25
`session_count = 3`	none	literal	`session_count`	3
`planned_minutes = ...`	?	?	?	?
`break_count = ...`	?	?	?	?
`break_minutes = ...`	?	?	?	?
`active_minutes = ...`	?	?	?	?
`completion_ratio = ...`	?	?	?	?

Then write the three output lines. Run the program only after the table is complete.

Show the completed values

- `planned_minutes` is `25 * 3`, or `75`.
- `break_count` is `3 - 1`, or `2`.
- `break_minutes` is `2 * 5`, or `10`.
- `active_minutes` is `75 - 10`, or `65`.

- `completion_ratio` is `65 / 75`, approximately `0.8667`.
- `round(completion_ratio, 2)` returns `0.87`.

The output is:

```text
Planned: 75
Active: 65
Ratio: 0.87
```

Modify one dependency at a time

1. Change `session_count` from `3` to `4`.
2. Predict which names will receive different values.
3. Run and compare.
4. Restore `3`.
5. Change the break length from `5` to `10`.
6. Again identify every downstream value that changes.

This is a **dependency trace**: a changed input affects expressions that read it, then expressions that read those results.

8. Repair execution-order mistakes

Output happens before the calculation

```python
price = 8
quantity = 4
print(total)
total = price * quantity
```

Move the display below the assignment to `total`.

A derived value is calculated from an old input

```python
hourly_rate = 20
hours = 3
pay = hourly_rate * hours

hours = 5
print(pay)
```

If `5` is the intended final input, calculate `pay` after that assignment.

Parentheses change the requirement

A shop intends to add a fixed delivery fee after applying a discount:

```python
subtotal = 80
delivery_fee = 10
discount_rate = 0.25
```

PYTHON (CONTINUED)

```
final_total = subtotal + delivery_fee * (1 - discount_rate)
print(final_total)
```

The current grouping discounts only the fee. A clearer version is:

PYTHON

```
discounted_subtotal = subtotal * (1 - discount_rate)
final_total = discounted_subtotal + delivery_fee
print(final_total)
```

This is not merely a precedence exercise. The correct expression depends on the business rule. State the rule in words before choosing parentheses.

One line hides too much

PYTHON

```
print(round((50 * 4 - 20) / 60, 2))
```

The code is valid, but its meaning is invisible. Give names to the meaningful stages. One possible story is four planned sessions, twenty break minutes, and a conversion from minutes to hours.

Show a traceable version

PYTHON

```
session_minutes = 50
session_count = 4
break_minutes = 20

planned_minutes = session_minutes * session_count
active_minutes = planned_minutes - break_minutes
active_hours = active_minutes / 60
rounded_hours = round(active_hours, 2)

print(rounded_hours)
```

9. Build a transparent event budget

Create a small budget calculator using these facts:

- room rental: `120`;
- materials per learner: `8`;
- learner count: `15`;
- refreshments: `45`;
- sponsor contribution: `100`.

Required names:

TEXT

```
room_cost
materials_per_learner
learner_count
refreshment_cost
sponsor_contribution
materials_total
```

TEXT (CONTINUED)

```
gross_cost
amount_to_raise
```

Requirements:

1. Calculate `materials_total`.
2. Calculate `gross_cost` from room, materials, and refreshments.
3. Subtract the sponsor contribution to calculate `amount_to_raise`.
4. Print labels with the three calculated results.
5. Use intermediate names rather than repeating a long expression.
6. Add parentheses only where they clarify grouping.
7. Make a prediction table before running.
8. Change the learner count to `20` and identify every downstream result that should change.

Progress checks:

PYTHON

```
assert materials_total == materials_per_learner * learner_count
assert gross_cost == room_cost + materials_total + refreshment_cost
assert amount_to_raise == gross_cost - sponsor_contribution
```

These assertions are supplied checks: each comparison must produce `True`. Testing receives a complete treatment in Unit 13.

Hint: calculate in dependency order

Assign the five input facts first. `materials_total` depends on two inputs. `gross_cost` depends on `materials_total`. `amount_to_raise` depends on `gross_cost`. Place each assignment after the values it reads.

Show one complete solution

PYTHON

```
room_cost = 120
materials_per_learner = 8
learner_count = 15
refreshment_cost = 45
sponsor_contribution = 100

materials_total = materials_per_learner * learner_count
gross_cost = room_cost + materials_total + refreshment_cost
amount_to_raise = gross_cost - sponsor_contribution

print("Materials:", materials_total)
print("Gross cost:", gross_cost)
print("Amount to raise:", amount_to_raise)

assert materials_total == 120
assert gross_cost == 285
assert amount_to_raise == 185
```

10. Check your execution trace

Read a program in execution order

Use the stated program order and grouping.

1. What is the first calculation in result = 60 - (10 + 5)?

- **A.** 10 + 5
- **B.** 60 - 10
- **C.** Bind result to 60
- **D.** Print 5

2. What is the strongest reason to store discount = subtotal * rate?

- **A.** Python cannot calculate it inline
- **B.** The name exposes a meaningful stage that can be inspected
- **C.** Every expression must be assigned twice
- **D.** It prevents subtotal from having a value

3. In difference = abs(35 - 50), what value is passed to abs()?

- **A.** 15
- **B.** 50
- **C.** -15
- **D.** difference

4. Two parenthesized formulas produce different totals. What should determine which one is correct?

- **A.** The shorter formula
- **B.** The one with more operators
- **C.** Whichever was run first
- **D.** The stated rule the program must represent

Write your choices before checking the answer key at the end of this book.

Key points

Key points

- An expression is evaluated to produce a value.
- Operators combine or compare operands.
- Parentheses make grouping explicit and can change a result.
- Function-call arguments are evaluated before the call produces its result or performs its action.
- Statements normally run from top to bottom.
- Calculating, storing, and displaying are different actions.

- Intermediate names make meaningful stages visible.
- A clean top-to-bottom run exposes hidden notebook dependencies. :::

References

- Python tutorial: numbers and arithmetic
- Python language reference: expressions
- Python language reference: simple statements
- Python documentation: built-in functions

Python's Words and Symbols

Online lesson and runnable notebook

Learn to identify Python's keywords, identifiers, literals, operators, and delimiters, then use that vocabulary to read and repair unfamiliar lines.

- **Level:** Beginner
- **Estimated time:** 2.5–3.5 hours
- **You will learn:** Identify the roles of Python’s words and symbols, create valid readable names, and diagnose token-level mistakes.
- **Practice in:** Google Colab, JupyterLab, or a local editor

At first, Python punctuation can look decorative. It is not. A quote can turn digits into text. One extra equals sign can turn assignment into comparison. A colon can announce a block. Parentheses can group a calculation or call a tool.

Learning the vocabulary lets you read unfamiliar code without pretending you already understand its complete behavior.

Questions you will answer

1 Which pieces of a line are names, literals, keywords, operators, or delimiters?

2 Why are some words unavailable as variable names?

3 Which naming mistakes stop Python, and which merely make code harder to read?

4 How does context change the role of a symbol? :::

1. Take apart one line of Python

Consider:

```python
total_minutes = session_count * 25
```

Python does not see one long command. It recognizes smaller pieces:

Piece	Role	Meaning here
`total_minutes`	identifier	the name that will receive a value
`=`	operator	assignment
`session_count`	identifier	a name whose value will be looked up
`*`	operator	multiplication
`25`	literal	an integer value written directly

A smallest meaningful piece recognized by Python’s tokenizer is called a **token**. You do not need to operate a tokenizer to read code, but the term helps us discuss the pieces precisely.

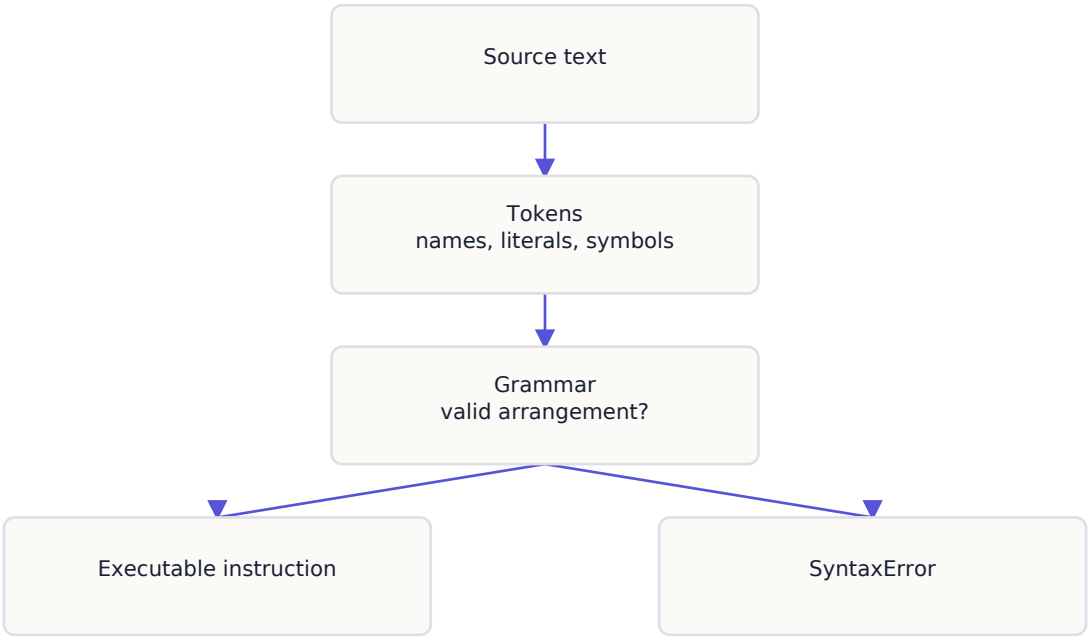
Another line has different token roles:

```python
print("Total:", total_minutes)
```

Piece	Role	Meaning here
`print`	identifier	the name of a built-in function
`(`	delimiter	begins the call's argument list
`"Total:"`	literal	a text value
`,`	delimiter	separates two arguments
`total_minutes`	identifier	a value to look up
`)`	delimiter	ends the argument list

Whitespace separates some tokens and helps people see the structure. It is not included in the table as an executable action.

Python first recognizes pieces, then checks whether their arrangement follows the language grammar.



Grammar / valid arrangement? → Executable instruction: yes

Grammar / valid arrangement? → SyntaxError: no

Label a line yourself

Copy this line and write each token in a table:

```python
is_ready = completed_tasks >= 3
```

You should find:

- two identifiers: `is_ready`, `completed_tasks`;
- two operators: `=`, `>=`;
- one integer literal: `3`.

The underscore is part of each identifier. The two characters `>` and `=` combine into the single comparison operator `>=`.

2. Identifiers are names that follow spelling rules

An **identifier** is the grammatical form Python uses for names. These are valid identifiers:

```python
course
course_name
session2
_private_note
café
```

Python permits Unicode letters, which is why `café` is valid. This course uses simple ASCII names in most code because they are easier to type consistently across keyboards and tools.

These are not valid identifiers:

```text
2sessions
course-name
course name
session.minutes
```

Why:

- `2sessions` starts with a digit;
- `course-name` is read as `course - name`, not one identifier;
- the space in `course name` separates two identifiers;
- the dot in `session.minutes` separates an object expression and an attribute name; the complete text is not one identifier.

The practical rules are:

1. Start with a letter or underscore.
2. Continue with letters, digits, or underscores.
3. Do not include spaces, hyphens, dots, quotation marks, or other punctuation.
4. Match capitalization exactly.
5. Do not use a reserved keyword.

Valid syntax can still be poor naming

All these assignments are valid:

```python
x = 45
thing = 10
this_is_the_number_of_minutes_remaining_after_the_break = 35
```

Validity only tells you Python can read the name. Readability asks whether a person can understand it.

Prefer:

```python
session_minutes = 45
break_minutes = 10
focused_minutes = 35
```

Good names are specific enough to explain the role and short enough to scan.

Snake case joins multiple words

Python code normally writes variable and function names in **snake case**: lowercase words separated by underscores.

```python
learner_count = 18
materials_per_learner = 4
materials_total = learner_count * materials_per_learner
```

`learner_count` is one token, not two. The underscore belongs to the identifier.

Names beginning with one or two underscores can have special conventions. Beginners should not invent such names without a reason. `_temporary` is legal, but `temporary` is clearer unless a specific convention calls for the underscore.

3. Capitalization and spelling must match exactly

Python is case-sensitive:

```python
course = "Python Foundations"
Course = "Scientific Python"

print(course)
print(Course)
```

The two names have separate bindings.

This example fails at runtime:

```python
session_minutes = 45
print(Session_minutes)
```

The first character differs. Python raises `NameError` because `Session_minutes` was never assigned.

Common naming mismatches include:

Assigned	Read later	Difference
`ticket_count`	`tickets_count`	singular versus plural

Assigned	Read later	Difference
`total_cost`	`totalcost`	missing underscore
`course_name`	`course_Name`	capitalization
`focused_minutes`	`focus_minutes`	different word form

When you see `NameError`, copy the missing name from the error and compare it character by character with the assignments above.

Repair a naming set

Each line below intends to use a snake-case name. Rewrite it as one valid assignment:

```text
1st_session = 25
break-time = 5
course title = "Python"
total.minutes = 30
```

One possible repair:

```python
first_session = 25
break_time = 5
course_title = "Python"
total_minutes = 30
```

The exact words can differ. The repaired name must follow the identifier rules and communicate the intended role.

Check your understanding

Tokens and identifiers

Choose the answer that follows Python’s lexical rules.

1. Which text is one valid Python identifier?

- **A.** session_2
- **B.** 2_session
- **C.** session-time
- **D.** session time

2. Which name follows the usual snake-case style?

- **A.** FocusedMinutes
- **B.** focused_minutes
- **C.** focused-minutes
- **D.** FOCUSED MINUTES

3. course is assigned, but print(Course) raises NameError. Why?

- **A.** print requires lowercase arguments
- **B.** Text values cannot use names
- **C.** course and Course are different identifiers
- **D.** Every identifier must contain an underscore

4. In total = count * 5, which piece is a literal?

- **A.** total
- **B.** =
- **C.** *
- **D.** 5

Write your choices before checking the answer key at the end of this book.

4. Keywords belong to Python’s grammar

A **keyword** is a word reserved by Python for a grammatical role. Examples include:

```text
if      else      for      while
def     class     return   import
True    False     None     and
or      not       in       is
```

You will learn the behavior connected to these words in later units. For now, recognize that you cannot use them as ordinary names.

This is invalid:

```python
class = "Python Foundations"
```

Python sees `class` as the beginning of a class statement, not as a variable name. The rest of the line does not fit that grammar, so parsing stops.

This repair is valid and clearer:

```python
course_class = "Python Foundations"
```

Keywords are not commands by themselves

Writing a keyword without the structure it requires may be invalid:

```python
if
```

The word `if` begins a conditional statement. Python expects a condition, a colon, and an indented body. Recognizing the keyword tells you which grammatical shape to expect.

Check a word after making a prediction

Python's standard library provides the `keyword` module. This example uses `import` and a loop before their full lessons, so read it as a guided exploration:

```python
import keyword

print(keyword.iskeyword("for"))
print(keyword.iskeyword("total"))
print(keyword.iskeyword("return"))
print(keyword.iskeyword("course_name"))
```

Expected output:

```text
True
False
True
False
```

`keyword.iskeyword(...)` answers whether a text value is currently reserved by Python.

Try `"match"`, `"case"`, `"print"`, and `"None"`. Predict each result first. `print` is a built-in name, not a keyword; shadowing it is legal but unwise.

5. Built-in names are available tools, not grammar

Python starts with names such as `print`, `round`, `abs`, `min`, and `max` available. They are **built-ins**. Unlike keywords, they are ordinary names and can technically be rebound:

```python
print = 42
```

The assignment is valid, but the next call fails because `print` now refers to a number:

```python
print("Hello")
```

In a notebook, restart the runtime after accidentally shadowing a built-in. In a file, rename the variable and run the file again.

Compare the categories:

Category	Example	Can it be a new variable name?
Keyword	`for`	No
Built-in name	`print`	Yes, but usually should not
Identifier you create	`course_name`	Yes
Literal	`"course_name"`	No; this is a text value

Quotation marks are decisive. `course_name` asks Python to look up a name; `"course_name"` creates text containing those characters.

Diagnose four different lines

```python
course = "Python"
print(course)
print("course")
print
```

- Line 1 assigns text to `course`.
- Line 2 calls the built-in and displays the value bound to `course`: `Python`.
- Line 3 displays the literal text `course`.
- Line 4 is a name expression. In a notebook it may display a representation of the function object, but it does not call the function because there are no parentheses.

6. Operators tell Python what relationship to apply

Operators are tokens such as `+`, `*`, `=`, and `>=`. Some consist of more than one character.

Assignment and comparison

```python
minutes = 45
print(minutes == 45)
print(minutes != 30)
```

- `=` binds.
- `==` asks “are these equal?”
- `!=` asks “are these different?”

Do not insert spaces inside a multi-character operator:

```python
print(minutes > = 30)
```

Write `>=` as one token:

```python
print(minutes >= 30)
```

Arithmetic operators

```python
total = 8 + 4
difference = 8 - 4
product = 8 * 4
quotient = 8 / 4
remainder = 9 % 4
power = 2 ** 3
```

This lesson focuses on reading the symbols. Unit 2 covers numeric behavior, precision, floor division, and specialized numeric types.

Word operators

Not every operator is punctuation. Python has word operators such as `and`, `or`, `not`, `in`, and `is`. Because these words have grammatical roles, they are also reserved keywords.

```python
is_ready = True
has_time = True
can_start = is_ready and has_time
print(can_start)
```

Boolean reasoning belongs to Units 2 and 4. Here, identify `and` as an operator that combines two operand expressions.

Context gives a symbol its role

The `-` symbol can negate one value or subtract a second value:

```python
temperature = -5
change = 10 - 5
```

The first `-` is unary: it has one operand. The second is binary: it appears between two operands.

The `*` symbol means multiplication in an arithmetic expression:

```python
area = 6 * 4
```

Later, you will see `*` used in function parameters, argument unpacking, and sequence repetition. Do not assign a universal English word to a symbol without reading its context.

Check your understanding

Keywords, built-ins, and operators

Identify the role the token has in the shown context.

1. Why can course_class be a name while class cannot?

- **A.** class is a reserved keyword
- **B.** Names must contain underscores
- **C.** course_class is a built-in
- **D.** class contains too few letters

2. Which description of print is accurate?

- **A.** It is punctuation
- **B.** It is a built-in name that should normally not be reused
- **C.** It is a reserved keyword
- **D.** It is a string literal

3. Which token means greater than or equal to?

- **A.** > =

- **B.** =>
- **C.** >=
- **D.** ==>

4. What does print("course") display?

- **A.** The value bound to course
- **B.** A syntax error
- **C.** The print function
- **D.** The literal text course

Write your choices before checking the answer key at the end of this book.

7. Delimiters mark boundaries and relationships

Delimiters separate, group, or enclose parts of code.

Parentheses: grouping or calling

Grouping:

```python
total = (2 + 3) * 4
```

Calling:

```python
print(total)
```

In the first line, the opening parenthesis follows an operator and groups an expression. In the second, it follows the name `print` and begins an argument list. Context tells you the role.

Commas: separating items

```python
print("Total:", total)
```

The comma separates two arguments. It is not part of either value.

A missing comma can produce a confusing error or an unintended expression:

```python
print("Total:" total)
```

Repair:

```python
print("Total:", total)
```

Colons: announcing a body or separating parts

A colon after a header announces an indented block:

```python
if total >= 20:
    print("Large total")
```

The `if` behavior is a preview. Read only the shape for now:

1. `if` is a keyword.
2. `total >= 20` is the condition expression.
3. `:` finishes the header.
4. The indented call is the body.

Later, a colon can also separate keys from values in a dictionary or a start from a stop in a slice. The same token can have several grammatical contexts.

Dots: selecting an attribute

```python
course_name = "python foundations"
print(course_name.title())
```

The dot connects the value found through `course_name` with the attribute name `title`. The following parentheses call that method. String methods receive full coverage in Unit 2; here, recognize that `course_name.title` is not one identifier.

Brackets and braces: enclosed structures

You will soon see:

```python
scores = [80, 90, 75]
settings = {"theme": "dark"}
```

Brackets delimit a list; braces delimit a dictionary. Collections receive their own unit. For syntax reading, notice the opening delimiter, separated contents, and matching closing delimiter.

8. Whitespace can separate tokens or create blocks

These lines mean the same thing:

```python
total=price*quantity
total = price * quantity
```

The second follows normal style and is easier to scan.

Sometimes a space is necessary:

```python
notready = True
if notready:
    print("Waiting")
```

The code above is actually valid, but `notready` is one identifier. If the intention was to apply the `not` operator to a name `ready`, write:

```python
ready = False
if not ready:
    print("Waiting")
```

Token boundaries change meaning. `notready` and `not ready` are different code.

Inside an indented block, leading whitespace has a grammatical role:

```python
if True:
    print("Inside the block")

print("Outside the block")
```

The next lesson examines indentation in depth. Spaces around `=` are style; spaces at the left edge can determine block membership.

Do not remove all spaces mechanically

Python can ignore some spaces, requires others to separate words, and uses leading indentation for blocks. A formatter understands these contexts; blind search-and-replace does not. :::

9. Annotate an unfamiliar program

Read this guided preview:

```python
course = "Python Foundations"
completed_tasks = 3
target_tasks = 5
is_finished = completed_tasks >= target_tasks

if is_finished:
    message = "Target reached"
else:
    message = "Keep going"

print(course, message)
```

You do not need to master decisions yet. Annotate it using this checklist:

1. Circle each literal.
2. Underline every identifier.
3. Box each keyword.
4. Mark each operator.
5. Pair every opening and closing parenthesis.
6. Mark the commas and colons.
7. Draw a vertical line along each indentation level.

Show an annotation guide

- Literals: `"Python Foundations"`, `3`, `5`, `"Target reached"`, `"Keep going"`.
- Identifiers: `course`, `completed_tasks`, `target_tasks`, `is_finished`, `message`, `print`.

- Keywords: `if`, `else`.
- Operators: the four assignment `=` tokens and the comparison `>=`.
- Parentheses: the pair in `print(course, message)`.
- Comma: separates the two arguments to `print`.
- Colons: complete the `if` and `else` headers.
- Indentation: each assignment to `message` belongs to the block above it.

Change one category at a time

1. Change only a literal: set `completed_tasks` to `5`.
2. Change only an identifier: rename `target_tasks` consistently.
3. Change only an operator: compare with `>` instead of `>=`.
4. Change punctuation incorrectly on purpose: remove one colon.
5. Restore the colon and explain the error message.

The output changes for different reasons. Naming the changed token category helps you explain the cause.

10. Complete a symbol-repair clinic

Each item contains one focused problem. Predict whether it is a syntax error, runtime name error, or valid-but-confusing code before revealing the repair.

A digit begins the intended name

```python
2nd_session = 45
```

Repair:

```python
second_session = 45
```

A keyword is used as a name

```python
return = 30
```

Repair:

```python
return_minutes = 30
```

A built-in is shadowed

```python
max = 90
print(max)
```

The code is valid and displays `90`, but future calls to `max(...)` fail in this runtime. Prefer:

```python
maximum_score = 90
print(maximum_score)
```

A multi-character operator is split

```python
is_ready = completed_tasks > = target_tasks
```

Repair:

```python
is_ready = completed_tasks >= target_tasks
```

A lookup uses a string by accident

```python
course = "Python Foundations"
print("course")
```

The code is valid, but it displays `course`. Remove the quotation marks to look up the name:

```python
print(course)
```

A call is missing its closing delimiter

```python
print("Ready"
```

Repair:

```python
print("Ready")
```

11. Check your reading vocabulary

Read Python’s words and symbols

Choose the category or repair that matches the exact code.

1. What does the comma do in print("Total:", total)?

- **A.** It separates two call arguments
- **B.** It adds the two values
- **C.** It begins a comment
- **D.** It ends the function call

2. How should course_name.title be read?

- **A.** As one identifier containing a dot

- **B.** As attribute title selected from the course_name value
- **C.** As subtraction
- **D.** As a string literal

3. Why can parentheses have more than one explanation?

- **A.** Python randomly assigns punctuation
- **B.** Parentheses are comments
- **C.** Their role depends on context, such as grouping or calling
- **D.** Every pair performs multiplication

4. Which repair makes print("Total:" total) a two-argument call?

- **A.** Remove print
- **B.** Replace both values with keywords
- **C.** Add an equals sign
- **D.** Insert a comma between the arguments

Write your choices before checking the answer key at the end of this book.

Key points

Key points

- Python recognizes source code as tokens arranged according to grammar.
- Identifiers begin with a letter or underscore and continue with letters, digits, or underscores.
- Keywords are reserved; built-in names are reusable but should usually not be shadowed.
- Literals write values directly; unquoted identifiers request name lookup.
- Operators describe operations or relationships.
- Delimiters group and separate parts of code.
- A symbol’s role depends on its context.
- Whitespace may be style, a required token separator, or block indentation. :::

References

- Python language reference: lexical analysis
- Python standard library: keyword checking
- Python language reference: identifiers and keywords
- PEP 8: naming conventions

Indentation and Code Blocks

Online lesson and runnable notebook

Read Python's left edge as a structural map, build correctly nested blocks, and diagnose both indentation errors and valid code at the wrong level.

- **Level:** Beginner
- **Estimated time:** 3–4 hours, with natural stopping points after Sections 2, 6, and 8
- **You will learn:** Use indentation to group code blocks and debug indentation errors.
- **Practice in:** Google Colab, Jupyter, or VS Code

Questions you will answer

1. How does the left edge show which statements belong together?
2. Which headers require an indented body?
3. Why can wrong indentation change behavior without producing an error?
4. How can you repair invisible tabs, spaces, and mismatched levels? :::

1. A colon opens a block

A Python block normally starts with a **header line** ending in a colon. The body of the block is indented under that header. When the indentation returns to a previous level, the block is over.

Here is the shape:

```text
header:
    body line 1
    body line 2
outside again
```

You will see this shape many times:

```python
if condition:
    do_this()

for item in items:
    do_that(item)

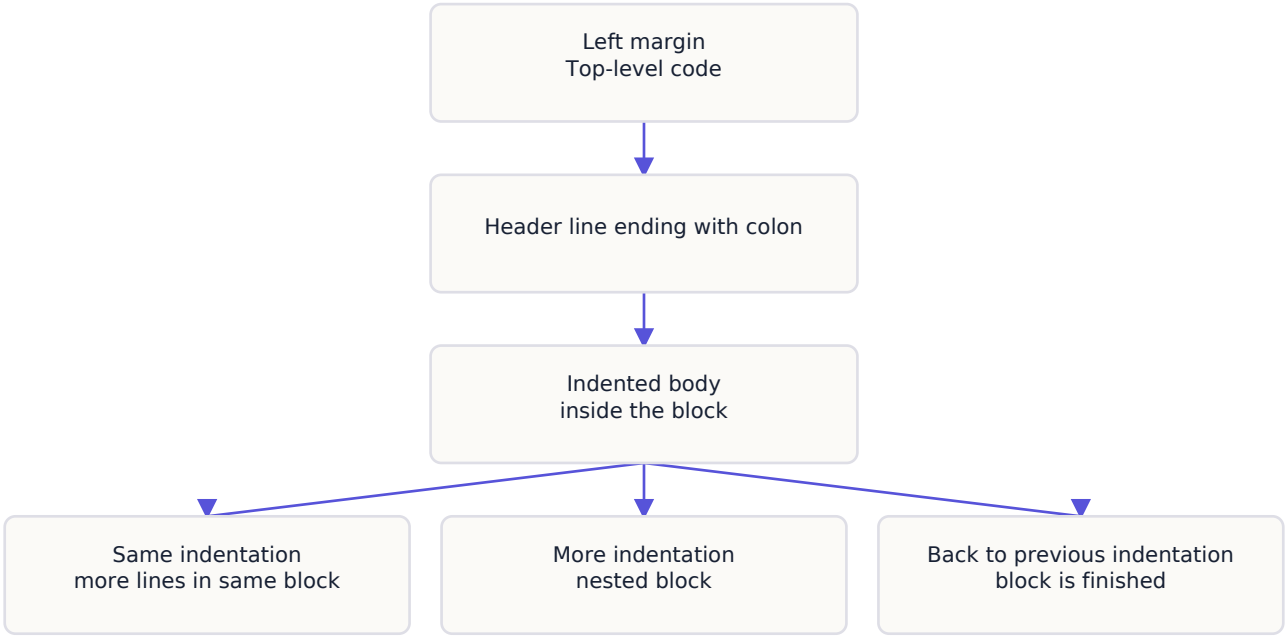
while still_working:
    keep_going()

def function_name():
    function_body()

class ClassName:
    class_body()
```

The words change, but the block shape stays the same. This is why indentation is a core concept, not a small formatting detail.

This diagram shows how indentation creates nested blocks.



Map your first block

```python
ready = True

if ready:
    print("Start")
    print("Work")

print("Finish")
```

Read it as a map:

- `ready = True` is top-level code.
- `if ready:` starts a conditional block.
- The two indented lines are inside that block.
- The final line is back at the top level.

Common issue: missing indented body

Broken code:

```python
ready = True

if ready:
print("Start")
```

The line after `if ready:` should be indented. Python cannot tell what belongs to the `if` statement, so it raises an `IndentationError`.

Fixed code:

```python
ready = True

if ready:
    print("Start")
```

Common issue: a block that is accidentally empty

Broken code:

```python
ready = True

if ready:

print("Start")
```

A blank line is allowed inside code, but it does not count as the body of the `if` statement. Python still expects an indented statement after the colon.

If you need a placeholder while planning code, use `pass`:

```python
ready = True

if ready:
    pass

print("Start later")
```

`pass` means "do nothing here on purpose." It is useful while sketching a program, but replace it with real code when the behavior is ready.

A preview of later Python

This lesson uses `if`, loops, functions, and classes because they all share the same visible block rule. You only need to read and repair their indentation shape here. Units 4, 5, and 11 teach their behavior in depth. :::

2. Keep `if`, `elif`, and `else` aligned

The `if`, `elif`, and `else` headers must line up with each other. Their bodies are indented one level deeper.

Map sibling branches

```python
temperature = 18

if temperature < 10:
    print("Wear a heavy coat")
elif temperature < 20:
    print("Wear a light jacket")
else:
    print("Short sleeves are fine")

print("Weather check complete")
```

The three headers are siblings:

```text
if temperature < 10:
elif temperature < 20:
else:
```

Each body is indented under its own header:

```text
    print(...)
```

Common issue: `else` indented under `if` body

Broken code:

```python
temperature = 18

if temperature < 10:
    print("Wear a heavy coat")
    else:
        print("Not that cold")
```

The `else` is too far to the right. Python reads it as if you are trying to put an `else` statement **inside** the `if` body, but `else` must match the `if`.

Fixed code:

```python
temperature = 18

if temperature < 10:
    print("Wear a heavy coat")
else:
    print("Not that cold")
```

Common issue: code that should be conditional but is outside

This code runs, but the indentation changes the meaning:

```python
logged_in = False

if logged_in:
    print("Welcome back")
print("Show private dashboard")
```

The dashboard line is not indented, so it runs even when `logged_in` is false. That is not a syntax error. It is a logic error.

Fixed code:

```python
logged_in = False

if logged_in:
    print("Welcome back")
    print("Show private dashboard")
```

Common mistake

Not every indentation bug creates an error message. Sometimes the code runs, but a line belongs to the wrong block. When output surprises you, inspect the left edge of the lines before changing conditions or values. :::

Check which lines belong to each branch

Indentation with conditionals checkpoint

Choose an answer, then check your work.

1. What usually comes after a Python line that ends with a colon?

- **A.** An indented block
- **B.** A closing curly brace
- **C.** A semicolon
- **D.** A line that must start with elif

2. Where should else line up in a simple if/else statement?

- **A.** Four spaces deeper than the if body
- **B.** With the matching if
- **C.** Inside the print statement
- **D.** At any indentation level

3. Why is an accidentally unindented line dangerous?

- **A.** Python always ignores it.
- **B.** It automatically becomes a comment.
- **C.** It may run outside the intended block without causing a syntax error.
- **D.** It can only happen inside classes.

Write your choices before checking the answer key at the end of this book.

3. Indent the work repeated by a `for` loop

A `for` loop repeats its indented body once for each item. Lines inside the loop are indented. Lines after the loop return to the previous indentation level.

Follow one loop body

PYTHON

```
names = ["Ada", "Grace", "Katherine"]

for name in names:
    print("Hello", name)
    print("This line is inside the loop")

print("All greetings sent")
```

Expected output:

```text
Hello Ada
This line is inside the loop
Hello Grace
This line is inside the loop
Hello Katherine
This line is inside the loop
All greetings sent
```

The first two `print` lines repeat because they are inside the loop. The final line prints once because it is outside the loop.

Common issue: the accumulator is inside the loop

This code is broken logically:

```python
scores = [10, 20, 30]

for score in scores:
    total = 0
    total = total + score

print(total)
```

It prints `30`, not `60`, because `total = 0` runs at the start of every loop trip. The accumulator resets again and again.

Fixed code:

```python
scores = [10, 20, 30]
total = 0

for score in scores:
    total = total + score

print(total)
```

The line `total = 0` belongs **before** the loop because it sets up the starting state once. The update belongs **inside** the loop because it should happen once per score.

Common issue: the loop body is too short

This code runs, but only the last value is summarized:

```python
prices = [4, 6, 8]

for price in prices:
    tax = price * 0.10
print("tax:", tax)
```

The final `print` is outside the loop. It prints once, after the loop, using the last value of `tax`.

If you want one line per price, indent the `print`:

```python
prices = [4, 6, 8]
```

PYTHON (CONTINUED)

```
for price in prices:
    tax = price * 0.10
    print("tax:", tax)
```

If you want only the last tax after the loop, leave it unindented, but name the choice clearly:

PYTHON

```
prices = [4, 6, 8]

for price in prices:
    last_tax = price * 0.10

print("last tax:", last_tax)
```

Move one line and predict the effect

Challenge

Move the final `print(total)` line inside the loop. Predict how many lines will print and what each line will show. :::

Show the explanation

PYTHON

```
scores = [10, 20, 30]
total = 0

for score in scores:
    total = total + score
    print(total)
```

Expected output:

TEXT

```
10
30
60
```

Inside the loop, `print(total)` runs after every update, so it shows the running total. Outside the loop, it would show only the final total.

4. Show which lines repeat in a `while` loop

A `while` loop repeats as long as its condition is true. Indentation decides what repeats. This is especially important because a misplaced update can create an infinite loop.

Follow the changing loop state

PYTHON

```
countdown = 3

while countdown > 0:
    print(countdown)
    countdown = countdown - 1

print("Go")
```

Expected output:

```text
3
2
1
Go
```

The update line `countdown = countdown - 1` is inside the loop, so the condition will eventually become false.

Common issue: update outside the loop

Broken code:

```python
countdown = 3

while countdown > 0:
    print(countdown)

countdown = countdown - 1
```

The update is outside the loop, so `countdown` stays `3` forever while the loop is running. In Colab, stop the cell if it keeps printing. Then move the update line into the loop body.

Fixed code:

```python
countdown = 3

while countdown > 0:
    print(countdown)
    countdown = countdown - 1
```

Common issue: `break` and `continue` at the wrong level

`break` and `continue` only make sense inside a loop.

Broken code:

```python
items = ["a", "stop", "c"]

for item in items:
    print(item)

    if item == "stop":
        print("found stop")

break
```

The `break` is at the top level, outside the `for` loop. Python raises a syntax error because there is no loop to break out of at that indentation level.

Fixed code:

```python
items = ["a", "stop", "c"]

for item in items:
    print(item)
```

```
PYTHON (CONTINUED)
    if item == "stop":
        print("found stop")
        break
```

The `break` is inside the `if`, and the `if` is inside the loop. That means `break` runs only when the loop reaches the stop item.

5. Put a function's body beneath `def`

A function definition has a header line and an indented body. Code inside the body runs only when the function is called. Code outside the body runs normally as the program reaches it.

Map a function body and its caller

```
PYTHON
def greet(name):
    message = f"Hello, {name}!"
    return message

print(greet("Ada"))
```

The function body is:

```
PYTHON
    message = f"Hello, {name}!"
    return message
```

The call is outside the function:

```
PYTHON
print(greet("Ada"))
```

Common issue: function body is not indented

Broken code:

```
PYTHON
def greet(name):
message = f"Hello, {name}!"
return message
```

Python raises `IndentationError` because a function definition needs an indented body.

Fixed code:

```
PYTHON
def greet(name):
    message = f"Hello, {name}!"
    return message
```

Common issue: `return` outside the function

Broken code:

```python
def add_one(number):
    result = number + 1

return result
```

The `return` line is not indented, so it is outside the function. Python raises `SyntaxError: 'return' outside function`.

Fixed code:

```python
def add_one(number):
    result = number + 1
    return result
```

Common issue: calling the function inside its own body by accident

This code is valid, but it hides the function call inside the function body:

```python
def greet(name):
    message = f"Hello, {name}!"
    print(message)
    greet("Ada")
```

The call `greet("Ada")` is indented inside `greet`, so calling `greet` would call itself again and again. That is recursion, and it is not what beginners usually intend here.

Move the call outside the function:

```python
def greet(name):
    message = f"Hello, {name}!"
    print(message)

greet("Ada")
```

Move `return` and trace the result

Challenge

Write a function named `is_passing(score)` that returns `True` when the score is at least `70` and `False` otherwise. Use indentation to put the `return` lines in the correct blocks. :::

Show one solution

```python
def is_passing(score):
    if score >= 70:
        return True
    else:
        return False

print(is_passing(85))
print(is_passing(50))
```

The `if` and `else` headers are inside the function. The `return` lines are inside the matching conditional bodies.

6. Put methods inside their class

A class body is indented under the `class` header. Methods are functions defined inside a class, so method definitions are indented under the class, and method bodies are indented under the method definition.

Map class, method, and method-body levels

```python
class Counter:
    def __init__(self):
        self.value = 0

    def increment(self):
        self.value = self.value + 1

    def report(self):
        return f"Counter value: {self.value}"

counter = Counter()
counter.increment()
print(counter.report())
```

There are three indentation levels:

```text
class Counter:                 # level 0
    def __init__(self):        # level 1, inside class
        self.value = 0         # level 2, inside method
```

The lines after the class return to the left margin:

```python
counter = Counter()
counter.increment()
print(counter.report())
```

Those lines are outside the class. They create and use an object.

Common issue: method is not inside the class

This code runs, but `increment` is not a method of `Counter`:

```python
class Counter:
    def __init__(self):
        self.value = 0

def increment(self):
    self.value = self.value + 1
```

The `increment` definition starts at the left margin, so it is a regular function, not a method inside `Counter`.

That difference becomes visible when you try to call it as a method:

```python
class Counter:
    def __init__(self):
        self.value = 0
```

PYTHON (CONTINUED)

```
def increment(self):
    self.value = self.value + 1

counter = Counter()
counter.increment()
```

Python creates a `Counter` object, but that object does not know about `increment`. The function named `increment` exists at the top level, beside the class, not inside the class. The likely error is:

TEXT

```
AttributeError: 'Counter' object has no attribute 'increment'
```

This error can be confusing because you can see a function named `increment` in the cell. The indentation is the clue: Python does not ask "is there a function nearby with this name?" It asks "does this object have a method with this name?" For `counter.increment()` to work, the `def increment(...)` line must be indented inside the `class Counter:` block.

Fixed code:

PYTHON

```
class Counter:
    def __init__(self):
        self.value = 0

    def increment(self):
        self.value = self.value + 1

counter = Counter()
counter.increment()
print(counter.value)
```

Expected output:

TEXT

```
1
```

Debugging routine

When a method call raises `AttributeError`, check two things before changing the method body:

1. Is the `def method_name(...)` line indented inside the class?
2. Did you spell the method name the same way in the class and in the call?
3. Is the method accidentally indented inside another method? :::

Common issue: method is inside another method by accident

There is a second version of the same mistake. Sometimes the method is not all the way outside the class. Instead, it is accidentally placed inside another method.

Broken code:

PYTHON

```
class Counter:
    def __init__(self):
        self.value = 0
```

PYTHON (CONTINUED)

```
        def increment(self):
            self.value = self.value + 1

counter = Counter()
counter.increment()
```

At first glance, this can look close to correct because `increment` is visually near the class. But count the indentation levels:

TEXT

```
class Counter:                  # level 0
    def __init__(self):         # level 1, method inside class
        self.value = 0          # level 2, inside __init__
        def increment(self):    # level 2, also inside __init__
```

The line `def increment(self):` lines up with `self.value = 0`, not with `def __init__(self):`. That means Python treats `increment` as a local function created while `__init__` is running. It is not a method stored on the `Counter` class, so this line:

PYTHON

```
counter.increment()
```

will likely raise:

TEXT

```
AttributeError: 'Counter' object has no attribute 'increment'
```

The fix is to move `def increment(self):` one indentation level to the left, so it lines up with the other method definition:

PYTHON

```
class Counter:
    def __init__(self):
        self.value = 0

    def increment(self):
        self.value = self.value + 1

counter = Counter()
counter.increment()
print(counter.value)
```

Expected output:

TEXT

```
1
```

How to spot this quickly

Inside a class, method definitions usually line up with each other. If `def increment(...)` is farther to the right than `def __init__(...)`, then `increment` is probably inside `__init__` by accident. :::

Common issue: method body is not indented enough

Broken code:

```python
class Counter:
    def __init__(self):
    self.value = 0
```

The assignment should be inside the method body. It needs one more indentation level than the method header.

Fixed code:

```python
class Counter:
    def __init__(self):
        self.value = 0
```

Common issue: object code accidentally inside the class

This code is valid in some cases but usually not what beginners intend:

```python
class Counter:
    def __init__(self):
        self.value = 0

    counter = Counter()
```

The line `counter = Counter()` is indented inside the class body. It tries to create a `Counter` while Python is still building the `Counter` class. Move object-creation code outside the class:

```python
class Counter:
    def __init__(self):
        self.value = 0

counter = Counter()
```

Common mistake

Classes often have two nested levels before any real behavior happens: method definitions inside the class, then method bodies inside each method. Count the levels before editing. :::

7. Fix tabs, spaces, and invisible mismatches

Python can technically read tabs, but the standard style is four spaces. Mixing tabs and spaces can produce errors that are hard to see because a tab may look like several spaces in your editor.

Common issue: inconsistent indentation

Broken code, with inconsistent whitespace:

```python
if True:
    print("four spaces")
    print("tab")
```

Python may raise `TabError: inconsistent use of tabs and spaces in indentation`. The fix is not to count invisible characters by hand. Use your editor to convert tabs to spaces.

In VS Code:

1. Open the command palette.
2. Search for **Convert Indentation to Spaces**.
3. Use four spaces per indent level.
4. Save and run again.

In Colab, avoid pressing Tab at the beginning of a line unless you know how the notebook handles it. If indentation looks strange, delete the whitespace at the start of the line and press Space four times.

Common issue: indentation width changes inside the same block

Broken code:

```
PYTHON
if True:
    print("line one")
  print("line two")
```

The second print line is indented two spaces, not four. Python sees it as a mismatched indentation level.

Fixed code:

```
PYTHON
if True:
    print("line one")
    print("line two")
```

The exact number of spaces matters less than consistency, but four spaces is the community standard and the style used throughout this course.

8. Read nested blocks along the left edge

A nested block is a block inside another block. You have already seen this with loops inside functions, `if` statements inside loops, and methods inside classes. Each nested level moves four more spaces to the right.

Trace three nested levels

```
PYTHON
def summarize_scores(scores):
    total = 0

    for score in scores:
        if score >= 70:
            print("passing score:", score)
        else:
            print("needs review:", score)

        total = total + score

    return total

print(summarize_scores([80, 55, 90]))
```

Indentation levels:

```text
def summarize_scores(scores):        # level 0
    total = 0                         # level 1, function body
    for score in scores:              # level 1, function body
        if score >= 70:               # level 2, loop body
            print(...)                # level 3, if body
        else:                         # level 2, matches if
            print(...)                # level 3, else body
        total = total + score         # level 2, loop body
    return total                      # level 1, function body
print(...)                            # level 0, outside function
```

This example is worth tracing slowly. Ask three questions for each line:

1. Is this line inside the function?
2. Is this line inside the loop?
3. Is this line inside the `if` or `else` body?

Move `return` into the loop and predict the effect

Challenge

Move `return total` four spaces to the right so it is inside the loop. Predict what changes before running.
:::

Show the explanation

If `return total` is inside the loop, the function returns during the first loop trip. That means it never processes the remaining scores. The code may run, but the logic is wrong.

Broken version:

```python
def summarize_scores(scores):
    total = 0

    for score in scores:
        total = total + score
        return total

print(summarize_scores([80, 55, 90]))
```

Expected output:

```text
80
```

Fixed version:

```python
def summarize_scores(scores):
    total = 0

    for score in scores:
        total = total + score

    return total

print(summarize_scores([80, 55, 90]))
```

Expected output:

```text
225
```

9. Diagnose indentation before changing program logic

Use this checklist when code looks correct but Python complains about spacing or runs lines at the wrong time.

Symptom	Likely cause	First thing to check
`IndentationError: expected an indented block`	A header line ending in `:` has no indented body	Look at the first real line after the colon
`IndentationError: unexpected indent`	A line is indented even though no block is open	Look at the previous non-blank line
`IndentationError: unindent does not match any outer indentation level`	A line returned to a column Python does not recognize	Check spaces/tabs and block levels
`TabError`	Tabs and spaces are mixed	Convert indentation to spaces
Code runs too often	A line is inside a loop by accident	Check whether it should be left of the loop body
Code runs too rarely	A line is outside a loop or conditional by accident	Check whether it should be indented under the header
`return outside function`	`return` is not inside a function body	Indent it under the correct `def`
`break outside loop`	`break` is not inside a loop body	Indent it under the correct `for` or `while`
Method missing from class	`def method(...)` starts at the left margin	Indent the method under `class`
Method missing from object	`def method(...)` is nested inside another method	Line up method definitions with each other

Mark every incorrect level before fixing it

Broken code:

```python
class StudyLog:
    def __init__(self):
        self.minutes = []

    def add(self, minutes):
        if minutes > 0:
            self.minutes.append(minutes)
        else:
        print("minutes must be positive")

    def total(self):
        total = 0
```

PYTHON (CONTINUED)

```
        for minutes in self.minutes:
        total = total + minutes
        return total

log = StudyLog()
log.add(20)
log.add(-5)
print(log.total())
```

Before fixing, identify the problem lines:

1. `print("minutes must be positive")` should be inside the `else` body.
2. `total = total + minutes` should be inside the `for` loop body.
3. `return total` should be inside the method but outside the loop.

Fixed code:

PYTHON

```
class StudyLog:
    def __init__(self):
        self.minutes = []

    def add(self, minutes):
        if minutes > 0:
            self.minutes.append(minutes)
        else:
            print("minutes must be positive")

    def total(self):
        total = 0
        for minutes in self.minutes:
            total = total + minutes
        return total

log = StudyLog()
log.add(20)
log.add(-5)
print(log.total())
```

Expected output:

TEXT

```
minutes must be positive
20
```

This example combines class, method, conditional, loop, and return indentation. It is a good final check because it uses several block levels at once.

10. Style rules that prevent indentation bugs

PEP 8, Python’s style guide, recommends four spaces per indentation level. This course follows that rule.

Good habits:

- Use four spaces for each block level.
- Configure your editor to insert spaces when you press Tab.
- Keep related lines at the same indentation level.

- Do not align code by hand with random numbers of spaces.
- Let formatters such as Ruff format handle mechanical spacing when possible.
- Fix indentation before debugging values, conditions, or algorithms.

Avoid this style:

```python
if True:
 print("one space")
 print("still works sometimes, but do not do this")
```

Prefer this style:

```python
if True:
    print("four spaces")
    print("clear block")
```

You will occasionally see compact one-line statements like this:

```python
if ready: print("Start")
```

Python allows it, but beginners should avoid it. Separate lines make the block structure visible:

```python
if ready:
    print("Start")
```

11. Map and repair a complete block structure

Challenge

Create a small `Notebook` class with three methods:

1. `__init__` creates an empty list named `notes`.
2. `add(note)` appends the note only if it is not an empty string.
3. `show()` loops over notes and prints each one.

Use an `if`, a `for` loop, a class, and methods. Then intentionally break one indentation level, run the cell, read the error, and fix it. :::

Show one possible solution

```python
class Notebook:
    def __init__(self):
        self.notes = []

    def add(self, note):
        if note != "":
            self.notes.append(note)
        else:
            print("empty note ignored")

    def show(self):
        for note in self.notes:
```

PYTHON (CONTINUED)

```
            print("-", note)

notebook = Notebook()
notebook.add("Practice indentation")
notebook.add("")
notebook.add("Run small examples")
notebook.show()
```

Expected output:

TEXT

```
empty note ignored
- Practice indentation
- Run small examples
```

The class body is indented under `class Notebook`. Each method is indented inside the class. Each method body is indented inside its `def` line. The `if` and `for` bodies are one level deeper inside their methods.

12. Check your understanding

Indentation in Python checkpoint

Choose an answer, then check your work.

1. Where should the body of a function be placed?

- **A.** At the left margin beside the def line
- **B.** Inside quotation marks
- **C.** Only after the function is called
- **D.** Indented under the def line

2. In a while countdown, where should the line that decreases the countdown usually go?

- **A.** Inside the while loop body
- **B.** Only after the loop has finished
- **C.** Inside a class but outside the loop
- **D.** In a comment

3. How can you tell that a function definition is a method inside a class?

- **A.** Its name always starts with a capital letter.
- **B.** Its def line is indented under the class header.
- **C.** It appears after all object creation code.
- **D.** It has no parameters.

4. In a class, def increment(…) is indented under def init(…) instead of lining up with it. What is the likely problem?

- **A.** increment becomes the class name.
- **B.** Python automatically moves increment to the correct indentation level.

- **C.** increment is a local function inside **init**, not a method on the object.
- **D.** The object will call increment before **init**.

5. What is the safest beginner habit for indentation whitespace?

- **A.** Use a random number of spaces as long as the code looks centered.
- **B.** Use tabs on some lines and spaces on others.
- **C.** Never indent after a colon.
- **D.** Use four spaces per indentation level and avoid mixing tabs and spaces.

6. Which indentation problem may not produce an error message?

- **A.** A line is outside a loop even though you meant it to be inside.
- **B.** A function header ends with a colon and has no body.
- **C.** Tabs and spaces are mixed in the same block.
- **D.** return appears outside every function.

Write your choices before checking the answer key at the end of this book.

Key points

Key points

- Python uses indentation to decide which lines belong to a block.
- A header line ending in `:` usually requires an indented body.
- `if`, `elif`, and `else` headers that belong together should line up.
- Loop setup often belongs before the loop; loop updates often belong inside the loop.
- Function bodies are indented under `def`; `return` must be inside the function.
- Methods are indented under `class`; method bodies are indented under method definitions.
- Method definitions inside the same class usually line up with each other.
- Use four spaces per indentation level and avoid mixing tabs and spaces.
- When debugging indentation, inspect block structure before changing program logic. :::

References

- Python Tutorial: control flow: https://docs.python.org/3/tutorial/controlflow.html
- Python Tutorial: defining functions: https://docs.python.org/3/tutorial/controlflow.html#defining-functions
- Python Tutorial: classes: https://docs.python.org/3/tutorial/classes.html
- PEP 8 indentation guidance: https://peps.python.org/pep-0008/#indentation

Writing Python Across Multiple Lines

Online lesson and runnable notebook

Break long Python statements across lines safely, distinguish continuation indentation from code blocks, and repair mismatched delimiters.

- **Level:** Beginner
- **Estimated time:** 2–3 hours
- **You will learn:** Format one Python statement across several readable lines without changing its result or confusing continuation with block structure.
- **Practice in:** Google Colab, JupyterLab, or a local editor

A long line forces readers to scroll, hides related parts, and makes missing punctuation harder to notice. Python lets one statement continue across several screen lines, but the opening and closing symbols must still describe one complete structure.

Questions you will answer

1 When does a newline end a statement?

2 How do parentheses, brackets, and braces keep a statement open?

3 Why is continuation indentation different from block indentation?

4 Which multiline styles are valid but unnecessarily fragile? :::

1. One statement can occupy several screen lines

A **physical line** is one line visible in the editor. A **logical line** is the complete instruction Python reads as one statement.

This assignment fits on one physical line:

```python
event_total = room_cost + materials_cost + refreshment_cost + equipment_cost
```

The same logical statement can use several physical lines:

```python
event_total = (
    room_cost
    + materials_cost
    + refreshment_cost
    + equipment_cost
)
```

The opening `(` tells Python the expression is not finished. Newlines inside the matching parentheses do not end the assignment.

Run a complete example:

```python
room_cost = 120
materials_cost = 80
refreshment_cost = 45
equipment_cost = 30

event_total = (
    room_cost
    + materials_cost
    + refreshment_cost
    + equipment_cost
)

print(event_total)
```

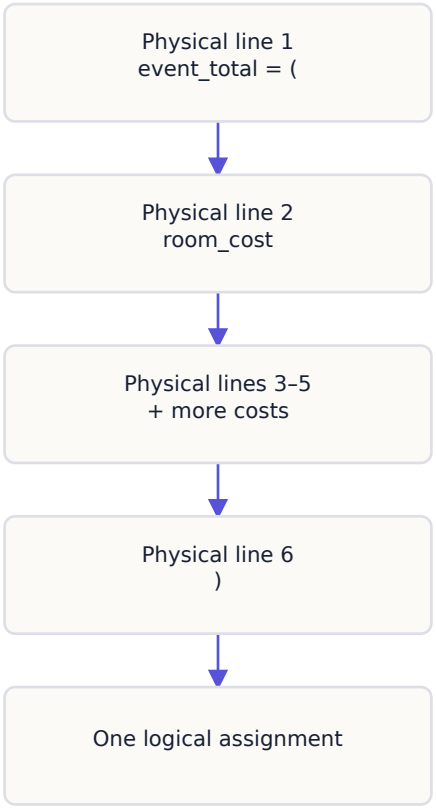
The output is `275`, exactly as it would be with a one-line expression.

What Python does on each line

1. The four input assignments run normally.
2. `event_total = (` starts an assignment whose right side is still open.
3. Python collects the following name and operator tokens.
4. The closing `)` completes the grouped expression.
5. Python evaluates the complete addition.
6. Assignment binds `event_total` to `275`.
7. The final call displays it.

The blank lines improve visual grouping. They do not create or end a block.

Several physical lines can form one logical statement while delimiters remain open.



2. Open delimiters permit safe continuation

Python supports implicit continuation inside three delimiter pairs:

Pair	Familiar use	Example
(and)	grouped expression or function call	`round(total, 2)`
[and]	list or indexing	`[25, 30, 45]`
{ and }	dictionary or set	`{"course": "Python"}`

Collections receive full coverage in Unit 3. Here, focus on the opening and closing shape.

Group a calculation

PYTHON
```
active_minutes = (
    session_count * session_minutes
    - break_count * break_minutes
)
```

The outer parentheses keep the expression open. The two physical calculation lines form one value.

Spread call arguments vertically

```python
print(
    "Course:",
    course_name,
    "Minutes:",
    active_minutes,
)
```

Each comma separates an argument. The final comma after `active_minutes` is a **trailing comma**. Python accepts it, and it makes adding or rearranging lines easier.

Lay out a collection as a preview

```python
session_lengths = [
    25,
    45,
    30,
]
```

The list still contains three values. The layout gives each item a clear line and leaves a trailing comma after the last item.

```python
print(session_lengths)
```

Expected output:

```text
[25, 45, 30]
```

A closing delimiter must match the opening one

This is invalid:

```python
session_lengths = [
    25,
    45,
    30,
)
```

The list opens with `[` but closes with `)`. Repair the final symbol to `]`.

Pair delimiters before reading the details

When a long statement looks confusing, first match every `(` with `)`, `[` with `]`, and `{` with `}`. Only then trace the values and operators inside. :::

3. Continuation indentation is not a code block

Compare a block:

```python
if session_minutes >= 30:
    message = "Focused session"
```

with a continued expression:

```python
message = (
    "Focused "
    + "session"
)
```

The indented block line belongs to the `if` header because the header ends in a colon. The indented string lines belong to an expression whose parenthesis is still open.

Both use visual indentation, but the grammar is different:

Question	Block	Continued statement
What starts it?	A header ending in `:`	An unclosed `(`, `[`, or `{`
What ends it?	Dedenting to an earlier level	The matching closing delimiter
Does indentation control behavior?	Yes	Usually no; delimiters control continuation
Typical style	Four spaces per block level	Align for readability

Continuation inside a block

The two forms can appear together:

```python
if session_minutes >= 30:
    message = (
        course_name
        + ": focused session"
    )
```

Read the left edge:

1. `if ...:` begins a block.
2. `message = (` is four spaces inside that block.
3. The continued expression is indented farther for readability.
4. The closing parenthesis lines up with the start of the continued expression.
5. Dedenting to the left margin would end the `if` block.

You do not need to master the condition yet. Your task is to distinguish the block level from the continuation layout.

Check your understanding

Logical lines and open delimiters

Identify what keeps each statement open.

1. What is a logical line in this lesson?

- **A.** One complete Python instruction, even if displayed across several lines

- **B.** Exactly one row in the editor
- **C.** Every line beginning with four spaces
- **D.** Only a line containing print

2. Which feature safely permits a calculation to continue after a newline?

- **A.** A comment on every line
- **B.** An unmatched opening parenthesis that will be closed later
- **C.** Changing every name
- **D.** A colon at the end of an assignment

3. A list begins with [. Which symbol must close that delimiter?

- **A.**)
- **B.** }
- **C.**]
- **D.** :

4. What primarily determines block membership?

- **A.** A trailing comma
- **B.** The width of the browser
- **C.** Parentheses around a value
- **D.** Indentation beneath a header ending in a colon

Write your choices before checking the answer key at the end of this book.

4. Choose a layout that exposes the structure

Python accepts several layouts inside parentheses. People still need a consistent visual pattern.

A compact call is clear when it is short:

```python
rounded_total = round(event_total, 2)
```

A vertical call is clearer when arguments need separate attention:

```python
print(
    "Event:",
    event_name,
    "Learners:",
    learner_count,
    "Total:",
    event_total,
)
```

Avoid a half-expanded shape:

```python
print("Event:", event_name,
    "Learners:", learner_count,
    "Total:", event_total)
```

It may run, but the argument boundaries are harder to scan and the indentation does not reveal a deliberate structure.

Prefer either one clear line or one argument per line:

```python
print(
    "Event:",
    event_name,
    "Learners:",
    learner_count,
    "Total:",
    event_total,
)
```

Break before operators consistently

This course uses operators at the beginning of continuation lines:

```python
net_total = (
    ticket_income
    + sponsor_income
    - room_cost
    - material_cost
)
```

The vertical operator column makes the additions and subtractions visible.

Do not alternate styles inside one expression:

```python
net_total = (
    ticket_income +
    sponsor_income
    - room_cost -
    material_cost
)
```

The code can be valid, but a consistent pattern is easier to review.

Use meaningful subexpressions

A very long continued expression can still hide its meaning:

```python
amount_due = (
    ticket_price * adult_count
    + ticket_price * 0.5 * child_count
    + 12
    - 20
)
```

Named stages may be better:

```python
adult_total = ticket_price * adult_count
child_total = ticket_price * 0.5 * child_count
booking_fee = 12
credit = 20

amount_due = (
    adult_total
    + child_total
    + booking_fee
    - credit
)
```

Multiline layout solves width. Intermediate names solve meaning. Use both when the program benefits from both.

5. Trailing commas support safer edits

Compare a vertical call without a trailing comma:

```python
print(
    course_name,
    focused_minutes
)
```

and with one:

```python
print(
    course_name,
    focused_minutes,
)
```

Both run. The trailing comma makes future editing simpler: a new argument can be added without modifying the previous line.

It also tells formatters that the structure may remain expanded. Formatting tools are covered later; the beginner habit is simple:

In a structure with one item per line, include a trailing comma after the last item when Python permits it.

A one-item tuple is a special case

This is a preview of Unit 3:

```python
one_score = (90,)
print(one_score)
```

The comma, not the parentheses alone, creates the one-item tuple. Compare:

```python
grouped_score = (90)
print(grouped_score)
```

`grouped_score` is just the number `90` grouped in parentheses.

Do not memorize all tuple behavior now. Remember that punctuation can carry meaning beyond visual layout.

6. Long text needs a deliberate strategy

This short text fits on one line:

```python
message = "Python Foundations begins with careful reading."
```

Adjacent string literals inside parentheses are joined by Python:

```python
message = (
    "Python Foundations begins with careful reading. "
    "Small experiments turn syntax rules into practical skill."
)

print(message)
```

The space after `reading.` is inside the first string. Without it, the words would touch.

For visible line breaks inside the text, use an escape sequence:

```python
message = "First session\nSecond session"
print(message)
```

Expected output:

```text
First session
Second session
```

Triple-quoted strings can contain physical newlines:

```python
message = """First session
Second session"""
print(message)
```

Triple-quoted strings also have a special role as docstrings when they appear in particular positions. The next lesson explains that difference.

Do not place a bare newline inside ordinary quotes

This is invalid:

```python
message = "First session
Second session"
```

An ordinary quoted string must close before the physical line ends. Use `\n`, adjacent literals in parentheses, or a triple-quoted string according to the desired value.

7. Avoid fragile continuation with backslashes

Python permits an explicit backslash at the end of a line:

```python
event_total = room_cost + materials_cost + \
    refreshment_cost
```

It works, but it is fragile. A space or comment after the backslash breaks the continuation, and editing delimiter-based code is usually safer.

Prefer:

```python
event_total = (
    room_cost
    + materials_cost
    + refreshment_cost
)
```

Do not write a comment after a continuation backslash:

```python
event_total = room_cost + materials_cost + \  # more costs below
    refreshment_cost
```

The backslash must be the final character on its physical line. The better repair is to use parentheses, not merely delete the comment.

Default to delimiters

Use implicit continuation inside `()`, `[]`, or `{}`. Treat backslash continuation as syntax you may need to recognize, not the normal style to write. :::

8. Put one simple statement on each line

Python permits semicolons between simple statements:

```python
course = "Python"; minutes = 45; print(course, minutes)
```

The line runs, but three actions are hidden on one physical line. Prefer:

```python
course = "Python"
minutes = 45
print(course, minutes)
```

A semicolon does not help with a complex statement that owns an indented body. This is invalid and unreadable:

```python
if minutes >= 30: message = "focused"; else: message = "short"
```

Use visible blocks:

```python
if minutes >= 30:
    message = "focused"
else:
    message = "short"
```

The decision behavior is still a preview. The syntax lesson is that blocks deserve their own indented lines.

Check your understanding

Readable multiline style

Choose the layout that preserves behavior and exposes structure.

1. Why is a trailing comma useful in a one-item-per-line call?

- **A.** It supports adding or rearranging arguments with smaller edits
- **B.** It turns every value into text
- **C.** It closes the parenthesis
- **D.** It starts a code block

2. A multiline expression is still hard to understand. What can improve its meaning?

- **A.** Remove all whitespace
- **B.** Give meaningful names to important intermediate results
- **C.** Replace every operator with a comma
- **D.** Put everything after a semicolon

3. How can an ordinary quoted string include a visible line break in its value?

- **A.** Place a raw newline before the closing quote
- **B.** Add a semicolon
- **C.** Use the escape sequence \n
- **D.** Indent the second word

4. What is the preferred repair for fragile backslash continuation?

- **A.** Add a comment after the backslash
- **B.** Put several statements on the line
- **C.** Remove all operators
- **D.** Wrap the continued expression in matching parentheses

Write your choices before checking the answer key at the end of this book.

9. Read delimiter-related errors from the outside in

Missing closing parenthesis

```python
total = (
    room_cost
    + materials_cost
```

Python may point at a later line or report that a parenthesis was never closed. The cause is the unmatched `(`.

Repair:

```python
total = (
    room_cost
    + materials_cost
)
```

Extra closing parenthesis

```python
total = room_cost + materials_cost)
```

There is no opening `(` for the final `)`. Remove it or add the intended opening group.

Missing comma between arguments

```python
print(
    "Course:"
    course_name,
)
```

The missing comma may be reported near `course_name`. Repair:

```python
print(
    "Course:",
    course_name,
)
```

A string silently joins when a comma was intended

This code is valid:

```python
print(
    "Course:"
    "Python Foundations",
)
```

Adjacent string literals join, so the output is:

```text
Course:Python Foundations
```

If the intention was two arguments, add a comma:

```python
print(
    "Course:",
    "Python Foundations",
)
```

Valid syntax can still hide a punctuation mistake. Compare behavior with the requirement, not only with the absence of an error.

A closing delimiter is too early

```python
total = (
    room_cost
    + materials_cost
) + refreshment_cost
```

This is valid. The final addition happens outside the parenthesized group. If the group is meant to show all event costs, move the closing delimiter:

```python
total = (
    room_cost
    + materials_cost
    + refreshment_cost
)
```

The numeric result is the same for addition, but the second layout communicates the intended grouping.

10. Reformat a registration summary

Start with this working but crowded program:

```python
event_name = "Community Python Workshop"
adult_count = 12
student_count = 8
adult_price = 20
student_price = 12
room_cost = 150
materials_cost = 65
income = adult_count * adult_price + student_count * student_price
balance = income - room_cost - materials_cost
print("Event:", event_name, "Adults:", adult_count, "Students:", student_count, "Income:", income, "Bal ↪
    ance:", balance)
```

Complete these tasks:

1. Predict `income` and `balance`.
2. Split the income calculation across multiple lines inside parentheses.
3. Introduce `adult_income` and `student_income` if they make the calculation easier to verify.
4. Format the `print()` call with one label or value per line.
5. Add trailing commas to the vertical argument list.

6 Keep one simple assignment per logical line.

7 Run before and after versions and confirm the values match.

8 Temporarily remove one closing delimiter, read the error, then restore it.

9 Add a new `sponsor_income = 100` input and update the balance without creating an unreadable line.

Progress checks before adding the sponsor:

```python
assert income == 336
assert balance == 121
```

Progress checks after adding the sponsor:

```python
assert sponsor_income == 100
assert balance == 221
```

Hint: separate meaning before arranging lines

Calculate adult and student income under separate names. Add those names inside a parenthesized `income` assignment. Then calculate `balance` from income and cost names. Formatting is easiest after the conceptual stages are clear.

Show one complete version

```python
event_name = "Community Python Workshop"
adult_count = 12
student_count = 8
adult_price = 20
student_price = 12
room_cost = 150
materials_cost = 65
sponsor_income = 100

adult_income = adult_count * adult_price
student_income = student_count * student_price
registration_income = (
    adult_income
    + student_income
)
total_income = (
    registration_income
    + sponsor_income
)
balance = (
    total_income
    - room_cost
    - materials_cost
)

print(
    "Event:",
    event_name,
    "Adults:",
    adult_count,
    "Students:",
    student_count,
    "Registration income:",
    registration_income,
    "Total income:",
    total_income,
```

PYTHON (CONTINUED)

```
    "Balance:",
    balance,
)

assert registration_income == 336
assert total_income == 436
assert balance == 221
```

11. Check your multiline reading

Trace code across physical lines

Treat matching delimiters as the boundary of each logical statement.

1. What ends an implicitly continued parenthesized expression?

- **A.** Its matching closing parenthesis
- **B.** Any newline
- **C.** Four leading spaces
- **D.** A comment anywhere below it

2. How should you verify that reformatting did not change a calculation?

- **A.** Count the physical lines
- **B.** Run the same behavior checks before and after
- **C.** Remove intermediate names
- **D.** Add a backslash to every line

3. Inside parentheses, what can happen to two adjacent string literals with no comma?

- **A.** They always raise SyntaxError
- **B.** They become a dictionary
- **C.** Python joins them into one string
- **D.** They start separate blocks

4. Why prefer one simple statement per line over semicolon-separated statements?

- **A.** Semicolons perform division
- **B.** Python cannot parse semicolons
- **C.** Newlines change every value
- **D.** Separate lines expose execution order and make edits easier

Write your choices before checking the answer key at the end of this book.

Key points

Key points

- A logical statement can span several physical editor lines.
- Matching parentheses, brackets, or braces permit implicit continuation.

- Block indentation follows a colon-terminated header; continuation indentation arranges tokens inside open delimiters.
- Consistent vertical layout and trailing commas make structures easier to edit.
- Intermediate names improve meaning when line breaks alone are not enough.
- Prefer delimiter-based continuation over backslashes.
- Prefer one simple statement per line over semicolon-packed code.
- Pair delimiters before diagnosing the details inside a long statement. :::

References

- Python language reference: line structure
- Python language reference: implicit line joining
- PEP 8: maximum line length and line breaks
- Python tutorial: first steps toward programming

Comments and Docstrings That Help

Online lesson and runnable notebook

Write comments and introductory docstrings that explain purpose, units, constraints, and decisions without merely translating the code.

- **Level:** Beginner
- **Estimated time:** 2–3 hours
- **You will learn:** Distinguish comments from docstrings and executable strings, then document decisions in a way that remains useful when code changes.
- **Practice in:** Google Colab, JupyterLab, or a local editor

Python code has two audiences. Python needs exact syntax. People need purpose, context, units, assumptions, and reasons. Clear names and structure should carry most of the explanation; comments and docstrings fill the gaps that code alone cannot express.

Questions you will answer

1. What information belongs in a comment rather than in a name?
2. How is a docstring different from a `#` comment?
3. When does Python make a docstring discoverable through `help()`?
4. Why can an outdated comment be more dangerous than no comment? :::

1. Comments speak to people reading the source

A `#` begins a comment outside a string. Python ignores the comment from that point to the end of the physical line.

```python
# Session lengths are measured in minutes.
session_minutes = 45
print(session_minutes)
```

The output is `45`. The comment does not create a value, perform assignment, or become part of the output.

A comment can follow code:

```python
break_minutes = 10  # Includes time to leave and return to the room.
```

Python performs the assignment and ignores the comment.

The `#` character inside a string is ordinary text:

```python
message = "Complete lesson #1"
print(message)
```

Expected output:

```text
Complete lesson #1
```

Quotation marks determine that `#1` belongs to the string rather than starting a comment.

Remove a comment and compare behavior

Run:

```python
# Keep the named stages so each cost can be checked.
room_cost = 120
materials_cost = 80
event_total = room_cost + materials_cost
print(event_total)
```

Then remove only the comment and rerun. The output stays `200`. A normal comment does not change program behavior.

This makes comments useful for explanation, but unsuitable for disabling important logic without careful review.

2. Prefer comments that explain why

This comment merely translates visible syntax:

```python
# Add room_cost and materials_cost.
event_total = room_cost + materials_cost
```

Anyone who recognizes `+` can read that. It adds maintenance work without adding knowledge.

This comment supplies context that the expression cannot:

```python
# The venue invoice includes equipment, so do not add it a second time.
event_total = room_cost + materials_cost
```

Useful comments often explain:

- why a non-obvious rule exists;
- which unit a number uses;
- where a requirement or constraint came from;
- why an apparently simpler approach is unsafe;
- what a temporary workaround is waiting for;
- what a surprising boundary means.

Put meaning in names first

Weak:

```python
m = 45  # minutes
b = 10  # break
r = m - b  # remaining focused time
```

Stronger:

```python
session_minutes = 45
break_minutes = 10
focused_minutes = session_minutes - break_minutes
```

The stronger version does not require three comments because the names carry the meaning.

A comment can still add a domain rule:

```python
# Break time is excluded because the report measures active instruction.
focused_minutes = session_minutes - break_minutes
```

Use this order

First improve the names and structure. Then add a comment only for important information that still is not visible. :::

3. Place comments where readers need them

A full-line comment introduces a decision

```python
ticket_income = 336

# The grant is paid directly to the venue, so it reduces costs rather than income.
venue_cost = 150 - 50
balance = ticket_income - venue_cost

print(balance)
```

The comment sits immediately above the decision it explains.

An inline comment labels a compact fact

```python
grace_period_seconds = 30  # Required by the device protocol.
```

Inline comments work best when both the code and explanation remain short. Avoid pushing a long paragraph to the right of a long expression.

A short comment can divide a small script

```python
# Inputs
room_cost = 120
materials_cost = 80

# Calculations
event_total = room_cost + materials_cost

# Output
print("Total:", event_total)
```

Section comments can help in a small educational script. In larger programs, functions and modules become better structural tools.

Too many comments interrupt reading

Avoid narrating every line:

```python
# Set the price to 12.
ticket_price = 12

# Set the count to 3.
ticket_count = 3

# Multiply price by count.
ticket_total = ticket_price * ticket_count

# Print the total.
print(ticket_total)
```

The code already tells that story. A single relevant comment may be enough:

```python
ticket_price = 12
ticket_count = 3

# Service fees are already included in the advertised ticket price.
ticket_total = ticket_price * ticket_count
print(ticket_total)
```

Check your understanding

Comments that add information

Choose the comment practice that helps the next reader most.

1. Which comment adds information not already visible in total = price * count?

- **A.** # The listed price already includes the service fee.
- **B.** # Multiply price and count.
- **C.** # Use an equals sign.
- **D.** # This is Python code.

2. What does # do inside the string "Lesson #1"?

- **A.** It deletes the rest of the program

- **B.** It becomes an ordinary character in the text value
- **C.** It begins a block
- **D.** It creates a docstring

3. What should usually happen before adding comments to explain x, y, and z?

- **A.** Put every statement on one line
- **B.** Replace values with keywords
- **C.** Rename them to express their roles
- **D.** Turn the assignments into strings

4. Where should a comment explaining a surprising calculation rule usually appear?

- **A.** In an unrelated later file
- **B.** Inside the numeric literal
- **C.** Only in the final output
- **D.** Immediately above or beside the relevant calculation

Write your choices before checking the answer key at the end of this book.

4. Comments must change when the code changes

An outdated comment lies:

```python
# Give a 10 percent discount.
discount_rate = 0.20
```

Python uses `0.20`; it ignores the conflicting comment. A reader may trust the wrong explanation and make a bad decision later.

Possible repairs depend on the real requirement:

```python
# Give a 20 percent discount during the community event.
discount_rate = 0.20
```

or:

```python
# Give the standard 10 percent discount.
discount_rate = 0.10
```

The absence of a syntax error does not decide which requirement is correct.

Review comments as part of every behavior change

Suppose the first version excludes breaks:

```python
# Report active time, excluding breaks.
reported_minutes = session_minutes - break_minutes
```

A later requirement includes the entire scheduled time:

```python
reported_minutes = session_minutes
```

The old comment must be removed or rewritten. Comments are part of the product even though Python does not execute them.

Temporary comments need an owner or condition

Vague:

```python
# TODO: fix this later
display_minutes = session_minutes
```

Useful:

```python
# TODO: include break_minutes after the schedule form collects it.
display_minutes = session_minutes
```

The second comment states what is missing and what condition allows the work to continue. A real project may also include an issue reference or owner.

Do not use `TODO` to avoid finishing a requirement that the current lesson or challenge expects.

5. A docstring documents a surrounding object

A **docstring** is a string literal placed as the first statement in a module, function, class, or method body. Python records it as that object's documentation.

At the top of a Python file or first code cell:

```python
"""Calculate and display a community workshop budget."""

room_cost = 120
materials_cost = 80
event_total = room_cost + materials_cost

print(event_total)
```

The opening string is the module docstring because it is the first statement.

A comment and module docstring have different jobs:

```python
"""Summarize one workshop budget."""

# The venue invoice already includes equipment rental.
room_cost = 120
materials_cost = 80
event_total = room_cost + materials_cost
```

- The docstring identifies the module's overall purpose.
- The comment explains one local domain decision.

- The assignments perform the calculation.

A random triple-quoted string is not automatically a docstring

```python
course = "Python Foundations"
"""This string appears after executable code."""
```

The triple-quoted text creates a string value, but it is not the module's docstring because another statement came first.

Triple quotes allow multiline strings. **Position**, not triple-quote spelling alone, gives a string its docstring role.

Inspect the current module docstring

In a notebook cell, run:

```python
"""Explore module documentation."""

print(__doc__)
```

Many notebook environments expose the cell or interactive module documentation differently, so the exact surrounding value can vary. The reliable language rule is that a module's first string statement becomes its `__doc__` value when the module is loaded.

A `.py` file makes this easiest to observe:

```python
"""Calculate a study-session summary."""

course = "Python Foundations"
minutes = 45

print(__doc__)
```

6. Function and class docstrings are previews

Functions are introduced fully in Unit 5. For now, read the placement:

```python
def focused_time(session_minutes, break_minutes):
    """Return active study minutes after subtracting a break."""
    return session_minutes - break_minutes
```

The levels are:

1. `def ...:` opens the function body.
2. The indented string is the first body statement, so it becomes the function docstring.
3. The indented `return` provides the function's behavior.

Move an assignment above the string:

```python
def focused_time(session_minutes, break_minutes):
    active_minutes = session_minutes - break_minutes
    """Return active study minutes after subtracting a break."""
    return active_minutes
```

The string is now an unused expression inside the function, not its docstring. The code is syntactically valid, but documentation discovery changes.

A class follows the same position rule:

```python
class StudySession:
    """Represent one planned study session."""

    pass
```

You do not need to design classes yet. Recognize the class header, indented body, first string, and placeholder `pass`.

Docstrings describe a public promise

A useful introductory docstring says what the module, function, or class provides. Detailed parameter formats, exceptions, examples, documentation toolchains, and publishing belong to Unit 15. :::

7. `help()` makes docstrings discoverable

Run:

```python
def focused_time(session_minutes, break_minutes):
    """Return active study minutes after subtracting a break."""
    return session_minutes - break_minutes


help(focused_time)
```

The output includes the function’s name, signature, and docstring.

You can also inspect the recorded text directly:

```python
print(focused_time.__doc__)
```

The dot selects the `__doc__` attribute. Attribute and method syntax receive more attention in later units; here it shows that docstrings are data Python preserves, not comments Python discards.

Compare with a comment-only function

```python
def focused_time(session_minutes, break_minutes):
    # Return active minutes after subtracting the break.
    return session_minutes - break_minutes


print(focused_time.__doc__)
```

The output is `None` because a `#` comment is not recorded as a docstring.

Check your understanding

Comments and docstrings

Pay attention to both syntax and position.

1. Which text becomes a module docstring?

- **A.** A string literal used as the module’s first statement
- **B.** Every # comment
- **C.** The last printed string
- **D.** Any string assigned to message

2. A string appears after an assignment inside a function. What is it?

- **A.** Always the function name
- **B.** An ordinary string expression, not the function docstring
- **C.** A syntax error
- **D.** A source comment

3. Which tool displays discoverable documentation for an object?

- **A.** round()
- **B.** print()
- **C.** help()
- **D.** abs()

4. Which statement is true?

- **A.** Comments and docstrings are identical tokens
- **B.** Every triple-quoted string starts a comment
- **C.** Python stores all # comments in **doc**
- **D.** Python ignores normal comments but records docstrings in special first-statement positions

Write your choices before checking the answer key at the end of this book.

8. Write a small docstring with a clear promise

For a module, begin with one direct sentence:

```python
"""Calculate attendance and cost summaries for one workshop."""
```

For a simple function preview:

```python
def material_cost(learner_count, cost_per_learner):
    """Return the total material cost for all learners."""
    return learner_count * cost_per_learner
```

Prefer a verb that describes the result or action:

- `Return ...`
- `Calculate ...`
- `Load ...`
- `Display ...`
- `Represent ...`

Avoid empty wording:

```python
def material_cost(learner_count, cost_per_learner):
    """This is the material_cost function."""
    return learner_count * cost_per_learner
```

The weak docstring repeats the name and says nothing about the result.

Document units and boundaries when they matter

```python
def focused_minutes(session_minutes, break_minutes):
    """Return active minutes; both inputs are measured in whole minutes."""
    return session_minutes - break_minutes
```

This is more useful because a caller can distinguish minutes from seconds and knows what “focused” excludes.

Do not promise validation or error handling the function does not provide. A docstring must match actual behavior.

9. Review good and harmful explanations

Example A

```python
# Multiply by 60.
duration_seconds = duration_minutes * 60
```

The comment repeats the operator but not the reason. Improve it:

```python
# The device API accepts seconds, while the form collects minutes.
duration_seconds = duration_minutes * 60
```

Example B

```python
# Never change this.
grace_period = 30
```

“Never” has no context. Improve it:

```python
# The access controller requires a 30-second grace period.
grace_period_seconds = 30
```

Example C

```python
"""Do stuff."""

room_cost = 120
materials_cost = 80
print(room_cost + materials_cost)
```

Improve the module docstring:

```python
"""Calculate the base cost of one community workshop."""

room_cost = 120
materials_cost = 80
print(room_cost + materials_cost)
```

Example D

```python
# Total is price times quantity.
total = price * quantity
```

Remove the comment. The names and expression already explain the line.

10. Prepare a program for a classmate

Start with:

```python
c = "Python Foundations"
m = 45
b = 10
f = m - b
print(c, f)
```

The behavior is small, but the program lacks context. Revise it so that:

1. a module docstring states the program’s purpose;
2. names express course, scheduled time, break time, and focused time;
3. one comment explains why break time is excluded;
4. no comment merely translates an assignment;
5. output labels identify both displayed values;
6. the result remains `35` focused minutes;
7. changing scheduled time to `60` requires changing only one input assignment.

Then give only the revised source to another person. Ask them:

- What unit is time measured in?
- Why is the break subtracted?
- Which value should they edit for a longer session?

- What output should appear for a 60-minute session?

If the code and documentation do not answer those questions, revise them.

Hint: let names carry the basic story

Use names such as `course_name`, `session_minutes`, `break_minutes`, and `focused_minutes`. Reserve the comment for the reporting rule that excludes the break.

Show one documented version

```python
"""Display the active study time for one planned course session."""

course_name = "Python Foundations"
session_minutes = 45
break_minutes = 10

# The study report measures active work, so scheduled breaks are excluded.
focused_minutes = session_minutes - break_minutes

print("Course:", course_name)
print("Focused minutes:", focused_minutes)

assert focused_minutes == 35
```

11. Find documentation drift

Run this first version:

```python
"""Calculate the cost after a standard 10 percent discount."""

subtotal = 200
discount_rate = 0.10
discount = subtotal * discount_rate
final_total = subtotal - discount

print(final_total)
```

Now imagine the event discount changes to 20 percent. Change only the numeric value and observe that the code runs while the docstring becomes false.

Your repair must update both behavior and documentation:

```python
"""Calculate the cost after the current event discount."""

subtotal = 200
discount_rate = 0.20
discount = subtotal * discount_rate
final_total = subtotal - discount

print(final_total)
```

The revised docstring avoids duplicating a value that already has a clear name. Another valid choice is to retain “20 percent” and commit to updating it whenever the rule changes. Decide which promise will be safer for the next reader.

12. Check the next-reader experience

Document code for its next reader

Choose the explanation that remains accurate and discoverable.

1. Which comment best explains duration_seconds = duration_minutes * 60?

- **A.** # The external API requires seconds, but the form collects minutes.
- **B.** # Multiply by 60.
- **C.** # Set a variable.
- **D.** # Python supports numbers.

2. A discount changes from 10% to 20%. What documentation should be reviewed?

- **A.** Only unrelated comments
- **B.** Any comment or docstring that promises the old rule
- **C.** No documentation because Python ignores it
- **D.** Only the filename

3. Where should a function docstring appear?

- **A.** After the final return
- **B.** Before the def line as a # comment
- **C.** As the first statement inside the function body
- **D.** Inside every print call

4. When should a comment usually be removed rather than expanded?

- **A.** When it records a surprising requirement
- **B.** When it states a required unit
- **C.** When it explains a workaround condition
- **D.** When it only restates clear code such as total = price * count

Write your choices before checking the answer key at the end of this book.

Key points

Key points

- A # comment is ignored by Python outside a string.
- Clear names and structure should explain ordinary mechanics.
- Useful comments record purpose, units, constraints, and non-obvious decisions.
- Comments must be updated when the behavior or requirement changes.
- A docstring is a string in the first statement position of a module, function, class, or method.
- Triple quotes alone do not make a string a docstring.
- `help()` and `.__doc__` expose recorded docstrings.

- Documentation should promise only behavior the code actually provides. :::

References

- Python tutorial: comments
- Python tutorial: documentation strings
- PEP 257: docstring conventions
- Python documentation: `help()`

Reading and Fixing Syntax Errors

Online lesson and runnable notebook

Read every part of a SyntaxError report, locate the broken grammatical rule, and repair common quote, delimiter, colon, name, and indentation mistakes.

- **Level:** Beginner
- **Estimated time:** 3–4 hours
- **You will learn:** Use the error type, message, line, caret, and nearby source to make one evidence-based syntax repair at a time.
- **Practice in:** Google Colab, JupyterLab, or a local editor

A syntax error is not Python refusing to help. It is Python showing where its attempt to read the program stopped. The message may not name the complete repair, but it narrows the search.

Questions you will answer

1. Which parts of an error report should you read before editing?
2. Why can the caret appear after the character that is actually missing?
3. How do syntax errors differ from runtime and logic failures?
4. What repair sequence avoids replacing one guess with several new mistakes? :::

1. Python checks grammar before running the program

When Python receives source code, it first needs to recognize tokens and arrange them according to the language grammar. If that parsing step fails, execution cannot begin.

This source is missing a closing quote:

```python
course = "Python Foundations
print(course)
```

Python reports a `SyntaxError`, often with wording similar to:

```text
  Cell In[1], line 1
    course = "Python Foundations
             ^
SyntaxError: unterminated string literal
```

The exact filename, cell label, caret width, and wording can vary by Python version and environment. The useful structure remains:

1. **Location:** the file or notebook cell and line number.
2. **Source:** the line Python was reading.
3. **Caret:** the point where Python detected that parsing could not continue.
4. **Type:** `SyntaxError`.

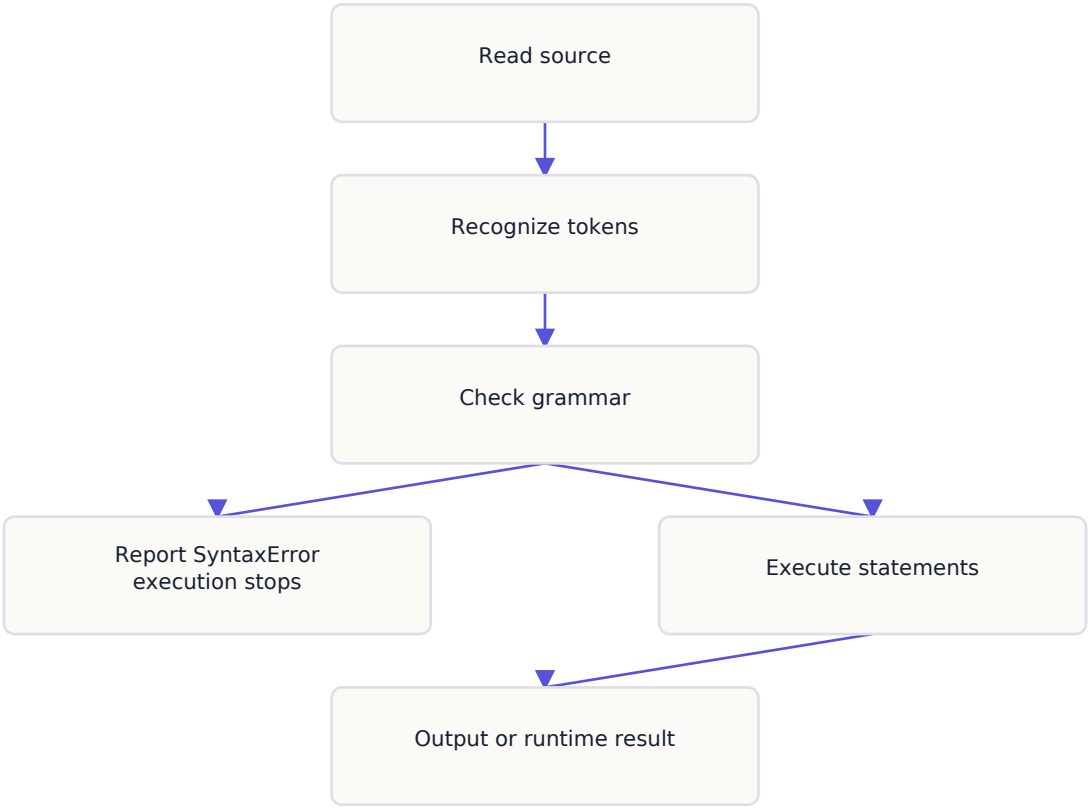
5 **Message:** a clue such as `unterminated string literal`.

Repair the missing quote:

```python
course = "Python Foundations"
print(course)
```

Now Python can parse the complete program and begin execution.

A syntax failure happens before ordinary statement execution.



Check grammar → Report SyntaxError / execution stops: invalid

Check grammar → Execute statements: valid

2. Read the report before touching the code

Use this five-part routine:

1 Read the final line: error type and message.

2 Read the named source line.

3 Locate the caret or highlighted range.

4 Inspect the line **before** it as well.

5 State one broken rule in words before making one edit.

Do not begin by rewriting the whole example. A small repair preserves the connection between cause and result.

The caret marks detection, not always cause

This invalid call is missing a comma:

```
PYTHON
print("Course:" course_name)
```

Python may underline `course_name` because adjacent tokens no longer fit a valid argument list. The missing character belongs **before** the highlighted name:

```
PYTHON
print("Course:", course_name)
```

Similarly, a delimiter left open on one line may cause Python to complain at a later line where the unfinished structure becomes impossible.

Inspect one line above

```
PYTHON
event_total = (
    room_cost
    + materials_cost

print(event_total)
```

The `print` line is where Python discovers the grouped expression never closed. The root cause is the missing `)` above it.

Repair:

```
PYTHON
event_total = (
    room_cost
    + materials_cost
)

print(event_total)
```

Say the rule before the edit

A useful diagnosis sounds like “The opening parenthesis has no matching closing parenthesis,” not “The red line looks wrong.” The first statement tells you which smallest edit to test. :::

3. Syntax, runtime, and logic failures are different

Not every broken program has invalid syntax.

Failure	Did parsing finish?	What happens?	Example
Syntax error	No	Python cannot start ordinary execution	missing quote
Runtime exception	Yes	Execution starts and then an operation fails	unknown name

Failure	Did parsing finish?	What happens?	Example
Logic error	Yes	Program runs but produces the wrong behavior	stale total

Syntax error

```
PYTHON
if minutes >= 30
    print("Focused")
```

The `if` header is missing its colon.

Runtime exception

```
PYTHON
print(minutes)
```

The syntax is valid. In a clean runtime, the line raises `NameError` because `minutes` has no binding.

Logic error

```
PYTHON
minutes = 45
break_minutes = 10
focused_minutes = minutes + break_minutes
print(focused_minutes)
```

The code parses and runs. If focused time should exclude the break, `+` represents the wrong rule. The result is incorrect without an exception.

Unit 8 develops all three categories and a complete debugging method. In this lesson, the distinction prevents you from searching for punctuation when the actual problem is missing state or wrong behavior.

Check your understanding

Read an error report

Classify the evidence before choosing a repair.

1. Which part of an error report gives the error type and summary message?

- **A.** The final line
- **B.** The notebook title
- **C.** The first comment
- **D.** The variable’s value

2. What does the caret in a SyntaxError report most reliably show?

- **A.** The only character that may be edited
- **B.** Where Python detected that parsing could not continue
- **C.** The value of every variable

- **D.** The line that executed successfully

3. print(missing_name) parses but the name has no binding. Which category is this?

- **A.** Syntax error
- **B.** Comment error
- **C.** Runtime exception
- **D.** Indentation style only

4. A program runs but adds a break when it should subtract it. Which category fits?

- **A.** Unclosed delimiter
- **B.** Invalid identifier
- **C.** Missing block body
- **D.** Logic error

Write your choices before checking the answer key at the end of this book.

4. Repair unclosed and mismatched quotes

One ordinary quote never closes

```python
message = "Ready to study
```

Repair:

```python
message = "Ready to study"
```

The closing quote has a different kind

```python
message = "Ready to study'
```

A string opened with `"` must close with `"`. Repair:

```python
message = "Ready to study"
```

Python allows single-quoted strings too:

```python
message = 'Ready to study'
```

Choose one matching pair.

An apostrophe ends a single-quoted string early

```python
message = 'Python's syntax is readable'
```

Python reads the quote after `Python` as the end of the string. Use double quotes around text containing an apostrophe:

```python
message = "Python's syntax is readable"
```

or escape the apostrophe:

```python
message = 'Python\'s syntax is readable'
```

The first version is often easier to scan.

Triple quotes must also match

```python
"""Summarize one study session."

course = "Python Foundations"
```

Three opening quotes need three closing quotes:

```python
"""Summarize one study session."""

course = "Python Foundations"
```

Repair practice

For each text below, predict the first place Python will stop:

```python
course = "Python Foundations'
message = "Start now
print(course, message)
```

Repair one quote problem, rerun, then repair the next. The first syntax error can hide later ones.

5. Pair every opening and closing delimiter

Missing closing parenthesis

```python
print("Minutes:", focused_minutes
```

Repair:

```python
print("Minutes:", focused_minutes)
```

Extra closing parenthesis

```python
focused_minutes = session_minutes - break_minutes)
```

Repair by removing the unmatched `)`:

```python
focused_minutes = session_minutes - break_minutes
```

Mismatched delimiter types

```python
session_lengths = [25, 45, 30)
```

Repair the closing delimiter:

```python
session_lengths = [25, 45, 30]
```

Nested delimiters need inside-out pairing

```python
print(round((45 - 10, 2))
```

Write the intended calls first:

```text
round(45 - 10, 2)
print(the rounded result)
```

Then pair them:

```python
print(round(45 - 10, 2))
```

Read from the inside out:

1. `(` after `print` opens the outer call.
2. `(` after `round` opens the inner call.
3. `)` after `2` closes `round`.
4. the final `)` closes `print`.

Let the editor help

Most editors highlight a delimiter and its partner when the cursor touches one. Use that feature. For a long expression:

1. place the cursor beside the first opening delimiter;
2. find its highlighted partner;
3. repeat for nested pairs;
4. only then inspect commas and operators.

This is faster and more reliable than counting punctuation in your head.

6. Put commas between neighboring items

Missing comma in a call

```python
print("Course:" course_name)
```

Repair:

```python
print("Course:", course_name)
```

Missing comma in a collection preview

```python
session_lengths = [25, 45 30]
```

Repair:

```python
session_lengths = [25, 45, 30]
```

Adjacent strings can hide the mistake

This is valid:

```python
labels = [
    "Course:"
    "Minutes:",
]
```

Python joins adjacent string literals, producing one list item: `"Course:Minutes:"`. If the intent is two items, add the comma:

```python
labels = [
    "Course:",
    "Minutes:",
]
```

No syntax error appears in the first version. The requirement and resulting data reveal the problem.

Commas are not decimal separators in Python source

```python
price = 12.50
```

A period forms the decimal literal. Writing `12,50` creates two comma-separated integer expressions in some contexts; it does not mean the decimal number twelve and a half.

Locale-aware number parsing belongs to later data-boundary lessons.

7. Finish headers with a colon and a body

Headers such as `if`, `for`, `while`, `def`, and `class` normally end with a colon and own an indented body.

Missing colon

```python
if session_minutes >= 30
    message = "Focused"
```

Repair:

```python
if session_minutes >= 30:
    message = "Focused"
```

Missing indented body

```python
if session_minutes >= 30:
message = "Focused"
```

Repair:

```python
if session_minutes >= 30:
    message = "Focused"
```

A blank line is not a body

```python
if session_minutes >= 30:

message = "Focused"
```

Use a real indented statement. If the behavior is not ready, `pass` is a valid placeholder:

```python
if session_minutes >= 30:
    pass

message = "Continue planning"
```

`else` must align with its partner

```python
if session_minutes >= 30:
    message = "Focused"
    else:
        message = "Short"
```

Repair:

```python
if session_minutes >= 30:
    message = "Focused"
else:
```

PYTHON (CONTINUED)
```
    message = "Short"
```

The indentation lesson provides many more block maps and repair cases.

Check your understanding

Common parser repairs

Choose the smallest edit that restores the intended grammar.

1. What is a clear repair for message = "Python’s ready"?

- **A.** Use double quotes around the complete text
- **B.** Remove the equals sign
- **C.** Indent the assignment
- **D.** Replace the apostrophe with a comma

2. A list opens with [but closes with). What is the direct repair?

- **A.** Add a colon
- **B.** Replace) with]
- **C.** Remove every comma
- **D.** Quote the variable name

3. Python highlights course_name in print("Course:" course_name). What is likely missing?

- **A.** A second print name
- **B.** Four leading spaces
- **C.** A comma before course_name
- **D.** A multiplication operator

4. Where should else align in a simple if/else statement?

- **A.** Inside the string
- **B.** Under the assignment in the if body
- **C.** At a random deeper level
- **D.** With its matching if header

Write your choices before checking the answer key at the end of this book.

8. Repair invalid and reserved names

Name begins with a digit

PYTHON
```
1st_session = 25
```

Repair:

```python
first_session = 25
```

Hyphen is interpreted as subtraction

```python
break-time = 10
```

This does not form one identifier. Repair with snake case:

```python
break_time = 10
```

Space separates identifiers

```python
course name = "Python Foundations"
```

Repair:

```python
course_name = "Python Foundations"
```

Keyword is reserved

```python
class = "Foundations"
```

Repair:

```python
course_level = "Foundations"
```

A built-in name is different

```python
print = "Foundations"
```

This is syntactically valid because `print` is a built-in, not a keyword. It is still a bad choice because it replaces access to the normal function in the current runtime.

Syntax repair asks “Can Python read it?” Code review also asks “Will people and later operations understand it safely?”

9. The first error can hide the next one

Try to diagnose this source without running:

```python
course name = "Python Foundations"
session_minutes = 45
break_minutes = 10

if session_minutes >= 30
```

PYTHON (CONTINUED)

```
focused_minutes = session_minutes - break_minutes

print("Focused minutes:" focused_minutes)
```

There are several errors:

1. `course name` contains a space.
2. The `if` header lacks a colon.
3. The block body lacks indentation.
4. The call lacks a comma.

Python cannot report every issue accurately in one pass. Its first report is based on the earliest point where parsing becomes impossible.

Use progressive repair:

1. Run the source.
2. Read the first report completely.
3. Repair one rule.
4. Run again.
5. Repeat until parsing succeeds.
6. Only then examine runtime and logic behavior.

A possible repaired version:

PYTHON

```
course_name = "Python Foundations"
session_minutes = 45
break_minutes = 10

if session_minutes >= 30:
    focused_minutes = session_minutes - break_minutes

print("Focused minutes:", focused_minutes)
```

This parses and runs for the current input. A later decisions lesson will ask what happens when `session_minutes` is below `30` and the assignment never runs. That is a runtime-path question, not a remaining syntax error.

10. Use a syntax-repair record

For one difficult error, fill this table before editing:

Field	Record
Error type and message	Copy the final line
Reported location	File/cell, line, and caret
Nearby structure	Quote, delimiter, comma, header, name, or indentation

Field	Record
Broken rule	State one grammatical mismatch
Smallest edit	Describe one change
Rerun result	New error or successful parse
Explanation	Why the edit fits the rule

Example:

Field	Record
Error type and message	`SyntaxError: '(' was never closed`
Reported location	line 3 at `print`
Nearby structure	grouped addition above
Broken rule	opening `(` has no closing `)`
Smallest edit	add `)` after `materials_cost`
Rerun result	program displays `200`
Explanation	the closing delimiter completes the right-side expression

The table prevents “I changed several things and it worked” from replacing an understood repair.

11. Run a mixed syntax-error clinic

The following four stages should be completed in separate cells. Keep a copy of each broken version and record the error before repairing it.

Stage A: quotes and names

```python
course title = "Python Foundations
print(course title)
```

Target:

```python
course_title = "Python Foundations"
print(course_title)
```

Stage B: delimiters and comma

```python
print(
    "Course:"
    course_title,
]
```

Target:

```python
print(
    "Course:",
    course_title,
)
```

Stage C: header and indentation

```python
session_minutes = 45

if session_minutes >= 30
message = "Focused session"
else:
    message = "Short session"
```

Target:

```python
session_minutes = 45

if session_minutes >= 30:
    message = "Focused session"
else:
    message = "Short session"
```

Stage D: multiline calculation

```python
focused_minutes = (
    session_minutes
    - 10

print("Focused:", focused_minutes)
```

Target:

```python
focused_minutes = (
    session_minutes
    - 10
)

print("Focused:", focused_minutes)
```

Final checks:

```python
assert course_title == "Python Foundations"
assert session_minutes == 45
assert message == "Focused session"
assert focused_minutes == 35
```

Hint: classify before repairing

- Stage A has one invalid identifier and one unclosed string.
- Stage B has a missing argument separator and mismatched closing delimiter.
- Stage C has a missing header delimiter and missing body indentation.

- Stage D has one open grouping delimiter.

Repair and rerun after each individual change.

12. Read editor feedback without surrendering judgment

Editors may underline syntax before you run the code. That early feedback is valuable, but treat it as a clue:

- bracket highlighting can expose missing pairs;
- visible whitespace can reveal tabs and inconsistent indentation;
- syntax coloring can show a string that accidentally continues too far;
- a formatter can refuse code it cannot parse;
- a linter can identify legal but suspicious patterns.

Still read Python’s actual error after running. Editor rules, Python version, and notebook parsing can differ. Your goal is to connect the source, the language rule, and the observed result.

13. Check your repair method

Repair syntax methodically

Choose the next action supported by the parser evidence.

1. Python points at print below a multiline calculation. What should you inspect first?

- **A.** Whether the expression above has unmatched delimiters
- **B.** The color theme
- **C.** Every file in the project
- **D.** The final numeric result

2. Why make one small repair before rerunning?

- **A.** Python permits only one edit per file
- **B.** It preserves the connection between the diagnosed rule and new evidence
- **C.** It guarantees no later error exists
- **D.** It turns runtime errors into comments

3. Why might Python report only one of several syntax problems?

- **A.** Python can parse every invalid program
- **B.** Later errors are comments
- **C.** Parsing stops at an early point where valid structure cannot continue
- **D.** Only indentation matters

4. The code now parses. What should happen next?

- **A.** Assume the behavior is correct
- **B.** Delete all checks

- **C.** Add random punctuation
- **D.** Run it and evaluate runtime behavior against the requirement

Write your choices before checking the answer key at the end of this book.

Key points

Key points

- A `SyntaxError` means Python could not finish parsing the source.
- Read the type, message, line, caret, and preceding structure before editing.
- The reported location is where failure was detected, not always where the missing character belongs.
- Pair quotes and delimiters, separate items with commas, complete headers with colons, and give blocks consistent indentation.
- Repair the earliest grammatical problem one change at a time.
- Syntax, runtime, and logic failures require different questions.
- A clean parse is the beginning of behavioral checking, not proof of correctness. :::

References

- Python tutorial: syntax errors
- Python language reference: lexical analysis
- Python built-in exception: `SyntaxError`
- Python tutorial: errors and exceptions

Unit Challenge: Repair and Explain a Broken Study Planner

Online lesson and runnable notebook

Repair a complete beginner Python program, prove its behavior with assertions, document its decisions, and explain every important line.

- **Level:** Python Foundations · Unit 1 challenge
- **Estimated time:** 90–120 minutes for a first careful attempt
- **Outcome assessed:** Read, repair, format, document, check, and explain a short Python program using Unit 1 syntax skills.
- **Evidence:** A clean program, passing assertions, one debugging record, and a line-by-line explanation

1. Your task: repair the study planner

A study-planner program was damaged while being copied from a message. Its purpose is simple, but Python cannot parse it. Repair it in stages until it:

- preserves the required names and input values;
- calculates focused time;
- chooses the correct session label;
- stores a progress comparison;
- displays a readable summary;
- begins with a useful module docstring;
- contains one comment explaining why the break is excluded;
- passes every supplied assertion; and
- still works after one requirement change.

The challenge intentionally combines independent and cascading mistakes. The first repair may reveal another error. Make one change, rerun, and record what the new result teaches you.

Work independently, but use the support deliberately

Attempt each stage before opening a hint. A hint should answer a specific question you can state, not replace reading the source and error report. :::

2. Understand the acceptance example

The repaired program starts with these facts:

Name	Value	Meaning
`course_title`	`"Python Foundations"`	course being studied

Name	Value	Meaning
`session_minutes`	`45`	scheduled session length
`break_minutes`	`10`	scheduled break
`completed_lessons`	`3`	lessons completed so far
`target_lessons`	`7`	current target

It must calculate:

Name	Expected value
`focused_minutes`	`35`
`session_label`	`"focused"`
`is_target_reached`	`False`

Expected output:

```text
Course: Python Foundations
Focused minutes: 35
Session: focused
Target reached: False
```

Constraints

1. Keep every required name exactly as shown.
2. Keep the initial input values unchanged until the controlled-change stage.
3. Use one module docstring as the first statement.
4. Keep the `if`/`else` block; do not replace the decision with a fixed string.
5. Write the focused-time calculation across multiple lines inside parentheses.
6. Use a comment to explain why the break is subtracted.
7. Do not edit expected assertion values merely to create a pass.
8. Do not use `compile()`, `exec()`, exception handling, functions, or classes.

Before editing, annotate the starter source on paper or in Markdown:

- circle literals;
- underline intended identifiers;
- box keywords;
- mark operators;
- pair delimiters where possible;
- draw a line along each intended indentation level.

3. Start from the contract

Copy this starter into one code cell. It is intentionally invalid.

```python
"""Do study stuff."""

course title = "Python Foundations"
session_minutes = 45
break-minutes = 10
completed_lessons = 3
target_lessons = 7

# Subtract the break.
focused_minutes = (
    session_minutes
    - break-minutes

if focused_minutes >= 30
session_label = "focused"
    else:
        session_label = "short"

is_target_reached = completed_lessons >= target_lessons

print("Course:" course title)
print(
    "Focused minutes:",
    focused_minutes,
]
print("Session:", session_label)
print("Target reached:", is_target_reached)
```

Do not replace the entire program with a new one. Repair the starter so each change remains connected to an observed problem.

Stage A: make every intended name valid

Inspect the two intended multiword names that do not follow identifier rules. Repair their definitions **and every later use**.

After your edit, state:

- why a space cannot occur inside one identifier;
- why a hyphen is read as an operator;
- why changing only the assignment would leave a later lookup inconsistent.

Run again. A different syntax report is progress.

Stage B: complete the multiline expression

Pair the opening parenthesis in the `focused_minutes` assignment. Keep the calculation spread across readable physical lines.

Replace the existing comment with one that explains the reporting rule rather than translating subtraction. For example, it should answer:

Why does this report exclude the break?

Run again and read the next report.

Stage C: repair the decision block

The `if` header, body, `else` header, and second body must form one visible structure.

Before editing, draw the intended shape:

```text
if condition:
    first body
else:
    second body
```

Then repair the missing delimiter and indentation. Do not change the comparison or label values.

Stage D: repair both output calls

One call is missing an argument separator. Another closes with the wrong delimiter. Repair the smallest relevant token in each call.

Once the program parses, compare its four output lines with the acceptance example. Valid syntax is not enough; spelling, order, and values must match.

Stage E: document the program’s promise

Replace `"""Do study stuff."""` with a one-sentence module docstring that describes the program’s actual purpose. Keep it as the first statement.

Review the focused-time comment. It should explain the decision to exclude the break, not say only “subtract the break.”

4. Build in small stages

Use this repair sequence:

1. Run the unchanged starter and preserve the first report.
2. Classify the nearby issue as name, quote, delimiter, comma, header, or indentation.
3. State one broken rule.
4. Make one smallest edit.
5. Rerun immediately.
6. Record whether the same error, a new error, or normal output appears.
7. Continue until the program parses.
8. Compare normal output with the acceptance example.
9. Run the assertions in groups.
10. Restart the runtime and run every challenge cell from the top.

A useful progress log is:

Run	Reported evidence	Rule you believe is broken	One edit	New evidence
1	Copy the error type, line, and message	One grammatical rule	One change	Same, new, or resolved
2	...	...	...	...

Do not fix by deleting the difficult feature

Removing the multiline expression, `if` block, documentation, or assertions would avoid practicing the unit skill. Repair the required structure instead. :::

5. Run progressive assertions

Run each group only after the repaired program reaches normal output.

Group 1: required input bindings

```python
assert course_title == "Python Foundations"
assert session_minutes == 45
assert break_minutes == 10
assert completed_lessons == 3
assert target_lessons == 7
```

If one fails, inspect the corresponding assignment. Do not edit the expected value.

Group 2: calculated values

```python
assert focused_minutes == session_minutes - break_minutes
assert focused_minutes == 35
assert is_target_reached == (completed_lessons >= target_lessons)
assert is_target_reached is False
```

The first and third checks restate relationships. The second and fourth verify the acceptance example.

Group 3: decision result

```python
assert session_label == "focused"
assert session_label in ("focused", "short")
```

The first check verifies the current input. The second protects the allowed vocabulary.

Group 4: required source qualities

Verify these by reading the source:

- the first statement is the module docstring;
- the focused-time comment explains *why* the break is excluded;
- the focused-time expression remains multiline and parenthesized;
- the two branch bodies use four spaces;

- `else` aligns with `if`;
- each `print()` call has matching parentheses and comma-separated arguments.

Assertions can verify values. They cannot prove that a comment is useful.

6. Use the hint ladder only when needed

Hint 1: classify the remaining problems

Work from top to bottom.

- The intended course name contains whitespace.
- The intended break name contains an operator.
- The focused-time grouping never closes.
- The decision header lacks its closing delimiter.
- The two branch headers and bodies do not form matching levels.
- The first output call lacks an argument separator.
- The multiline output call closes with a square bracket.

Fix only the first unresolved category, then rerun.

Hint 2: compare each broken shape with a valid shape

Valid identifier and continuation shapes:

```text
course_title = value
break_minutes = value

focused_minutes = (
    first value
    - second value
)
```

Valid decision shape:

```text
if comparison:
    assignment
else:
    assignment
```

Valid two-argument call:

```text
print("Label:", value)
```

Use these shapes to locate a mismatch; do not copy new behavior into the program.

Hint 3: expected repaired structure

The repaired executable statements, without the documentation wording, should have this order:

```text
five input assignments
focused-time assignment
if/else assigning session_label
comparison assigned to is_target_reached
four print calls
```

The focused calculation is `45 - 10`. The decision compares that result with `30`. The progress comparison checks `3 >= 7`.

If those relationships exist and the syntax is repaired, the acceptance values follow.

7. Keep debugging evidence

Choose one failure that required thought. Record:

Field	Your evidence
Input or source state	What exact code did you run?
Error type and message	What did Python report?
Reported location	Which line and token were highlighted?
Nearby cause	Was the actual cause on that line or earlier?
Hypothesis	Which one syntax rule explained it?
Controlled change	What one edit did you make?
Verified rerun	What changed after rerunning?
Explanation	Why did that edit resolve this failure?

A useful record contains exact source and observed text. “It was broken, then I fixed it” is not enough to reuse the method later.

8. Test a changed requirement

After all initial checks pass, change only:

```python
session_minutes = 25
completed_lessons = 7
```

Before running, predict:

- `focused_minutes`;
- `session_label`;
- `is_target_reached`;
- all four output lines.

The expected changed values are:

```python
assert focused_minutes == 15
assert session_label == "short"
assert is_target_reached is True
```

If the old labels remain, check whether a derived assignment was run before or after the changed input. Restart and run from the top.

Then restore the original input values and rerun every initial assertion. The program should support both examples without changing its decision logic.

9. Explain the repaired program

Write one sentence for each important line or group:

1. What value or name appears?
2. What lookup or calculation does Python perform?
3. What binding changes?
4. What block owns the line?
5. What output or stored result should follow?

Your explanation should distinguish:

- the docstring from the comment;
- assignment `=` from comparison `>=`;
- the multiline expression from the decision block;
- the `if` body from the `else` body;
- calculated values from displayed values.

10. Compare with a solution path

Reveal after your assertions pass or all three hints have been used

```python
"""Display a study-session summary and current lesson-target status."""

course_title = "Python Foundations"
session_minutes = 45
break_minutes = 10
completed_lessons = 3
target_lessons = 7

# The report measures active study, so the scheduled break is excluded.
focused_minutes = (
    session_minutes
    - break_minutes
)

if focused_minutes >= 30:
    session_label = "focused"
else:
    session_label = "short"

is_target_reached = completed_lessons >= target_lessons
```

PYTHON (CONTINUED)

```
print("Course:", course_title)
print(
    "Focused minutes:",
    focused_minutes,
)
print("Session:", session_label)
print("Target reached:", is_target_reached)
```

Passing checks:

PYTHON

```
assert course_title == "Python Foundations"
assert session_minutes == 45
assert break_minutes == 10
assert completed_lessons == 3
assert target_lessons == 7
assert focused_minutes == session_minutes - break_minutes
assert focused_minutes == 35
assert session_label == "focused"
assert is_target_reached == (completed_lessons >= target_lessons)
assert is_target_reached is False
```

Your docstring and comment may use different words if they accurately describe the same purpose and decision.

11. Check your understanding

Broken study planner review

Connect each answer to the source you repaired.

1. Why is course_title a valid repair for course title?

- **A.** The underscore joins the words inside one identifier
- **B.** Every name must contain title
- **C.** Quotation marks would perform assignment
- **D.** Spaces are valid only after print

2. What makes the focused_minutes assignment one logical statement across several lines?

- **A.** The comment above it
- **B.** Its matching opening and closing parentheses
- **C.** The later if keyword
- **D.** The output labels

3. Which visual relationship connects else to its if?

- **A.** else is inside the message string
- **B.** else is indented under the first body
- **C.** else aligns with the if header
- **D.** else appears after every print

4. Which source explanation should remain after the challenge?

- **A.** # Subtract with a minus sign.
- **B.** # Assign a variable.
- **C.** # This line has code.
- **D.** # The report excludes scheduled breaks because it measures active study.

Write your choices before checking the answer key at the end of this book.

12. Before you finish

Evidence rubric

Evidence	Ready to record when
Syntax	The program runs from a clean state with no syntax or indentation errors.
Values	All initial and changed-example assertions pass without editing expected values.
Structure	Names, delimiters, continuation, and block levels follow the required shapes.
Documentation	The first statement is a useful docstring and the comment explains a real decision.
Debugging	One record connects an exact failure to one hypothesis, edit, and rerun.
Explanation	Every important line can be described as a value, lookup, expression, statement, block, or output action.

This button stores a self-reported marker only in this browser. It does not submit work, grade the program, verify identity, or issue a certificate.

Key points

- Repair syntax from the earliest clear failure, one controlled edit at a time.
- Passing assertions verify relationships and values; source review verifies readability and documentation.
- A changed example tests whether the program represents a rule rather than one memorized output.
- A clean rerun and line-by-line explanation complete the challenge.

Unit 2: Numbers, Text, and Other Values

Online lesson and runnable notebook

Learn how Python represents counts, measurements, money, text, truth, missing information, and time before combining them in practical programs.

- **Level:** Python Foundations · Unit 2
- **Estimated time:** 22–34 hours including the challenge
- **Unit outcome:** Choose suitable values for counts, measurements, money, text, truth, missing information, and time; convert and format them deliberately; and explain important precision, parsing, and comparison boundaries.
- **Practice in:** Generated Colab notebooks or a local Python environment

1. The values behind everyday Python programs

Unit 1 taught you to read Python’s words and symbols, follow assignment, trace an expression, recognize blocks, and repair syntax. You can now ask a deeper question: **what kind of value should each name hold?**

Consider the beginning of a ticket for an imaginary observatory visit:

```python
visitor = "Mina"
group_size = 4
ticket_price = 7.50
has_evening_access = True
special_request = None
visit_date = "2026-08-14"
```

These values look simple, but each one makes a promise:

- `"Mina"` is text whose characters can be inspected and transformed.
- `4` is an exact count that can be divided into groups.
- `7.50` is a floating-point measurement-like number; exact money may need a different representation.
- `True` records a fact that can participate in a decision.
- `None` records that no special request was supplied.
- `"2026-08-14"` is still text. Python cannot safely do calendar arithmetic with it until the program parses it as a date.

This unit is about choosing those representations on purpose. A program becomes more trustworthy when its values match the questions it must answer.

2. One spelling can describe different values

The text `"12"`, integer `12`, and floating-point number `12.0` may display similarly, but Python gives them different operations and boundaries. The same is true of an empty string, zero, `False`, and `None`: all can signal “nothing” in casual conversation, yet they do not mean the same thing in a program.

You will learn to ask four questions whenever data enters a program:

1. **What does this value represent in the real task?**
2. **Which Python type supports the operations the task needs?**
3. **What information could conversion or rounding lose?**
4. **How should the program display the result without changing its meaning?**

Those questions connect every lesson in this unit.

3. Your path through the unit

Step	Lesson	What it adds
1	Values, Types, Conversion, and None	Inspect types, convert deliberately, recognize lost information, and distinguish missing data from zero or empty text.
2	Integers and Everyday Arithmetic	Model exact counts, complete groups, leftovers, powers, and signed quantities.
3	Floating-Point Numbers and Precision	Work with measurements, compare approximate results, and separate stored values from rounded displays.
4	Money, Exact Ratios, and Complex Numbers	Choose `Decimal`, `Fraction`, or `complex` when ordinary integer or float rules do not match the task.
5	Strings and Unicode Text	Inspect, slice, search, clean, transform, and combine human text.
6	Input, Output, and Clear Formatting	Turn console text into usable values and present results with readable formatting.
7	Booleans, Comparisons, and Truth	Produce named facts, combine them, and distinguish truthiness from explicit comparisons.
8	Dates, Times, and Durations	Parse calendar values, calculate durations, and recognize timezone-aware timestamps.
9	Unit Challenge	Repair a time-travel supply record and unlock its final clearance message.

The order matters. Conversion prepares values for arithmetic. Integer arithmetic makes the contrast with floats clearer. The two text lessons separate manipulating text from interacting with a person. Boolean facts then describe what is true about those values, and the date lesson adds calendar-aware behavior.

4. What Unit 1 already gave you

Bring these skills forward rather than waiting to feel that you have memorized everything:

- read an assignment from right to left: calculate the value, then bind the name;
- distinguish a value-producing expression from a statement that performs a step;
- follow function calls from their inner arguments outward;

- use parentheses to make calculation order explicit;
- recognize strings by their quotes and calls by their parentheses;
- read the final line of an error report and inspect the indicated source line;
- change one cause at a time when an output surprises you; and
- write comments that explain a decision rather than narrating obvious syntax.

You will see small `if` examples only when they clarify a boundary already introduced in Unit 1. Unit 4 will teach complete decision-making programs. Lists, loops, functions, exception handlers, and files also have their own later units; this unit will not require those concepts silently.

5. Set up one Unit 2 notebook

Open the generated Colab notebook or create a notebook named `unit-2-values.ipynb`. Use one heading for each lesson and keep four kinds of cells:

1. **Original example** — the unchanged code from the lesson.
2. **Prediction and output** — what you expected and what Python actually did.
3. **Your modification** — one meaningful change with a short explanation.
4. **Lab** — the complete artifact you can rerun from a clean state.

When an example is intentionally broken, retain the error long enough to record its type, message, and cause. Then repair a copy. A notebook containing only final successful code hides some of the most useful learning.

Reset before blaming Python

Notebook cells can be run out of order. When a name seems to have an impossible value, restart the runtime and run the relevant cells from top to bottom. A clean rerun separates the visible program from forgotten notebook state. :::

6. Choose a pace that leaves room for experiments

Most lessons in this unit need several sessions. A useful rhythm is:

- **Session A:** read and run the first half, including its checkpoint;
- **Session B:** finish the operations and boundary cases;
- **Session C:** complete the lab without revealing the solution immediately;
- **Session D:** correct quiz answers in your own words and rerun from a clean state.

Stop after a checkpoint or complete example rather than in the middle of a failure investigation. The estimated time includes typing, making mistakes, reading errors, and trying modifications. Those are part of the course, not delays outside it.

7. Before you continue

Run this short comparison and explain every printed value:

PYTHON

```
quantity_text = "4"
quantity = int(quantity_text)
missing_note = None

print(type(quantity_text).__name__)
print(type(quantity).__name__)
print(quantity + 1)
print(missing_note is None)
```

You are ready for the first lesson when you can identify which line creates text, which line creates an integer, why `quantity + 1` works, and why the final check does not confuse missing information with an empty note.

Continue to Values, Types, Conversion, and `None`.

Values, Types, Conversion, and None

Online lesson and runnable notebook

Inspect Python values, convert text and numbers deliberately, recognize information loss, and represent missing information with None.

- **Level:** Python Foundations · Unit 2
- **Estimated time:** 2.5–4 hours
- **You will learn:** Inspect value types, convert deliberately, identify lossy conversions, and distinguish missing, empty, and zero values.
- **Practice in:** Google Colab, JupyterLab, or a local editor

1. The text `"12"` and the number `12` behave differently

A concert form sends a ticket quantity as text:

```python
quantity_text = "12"
print(quantity_text + quantity_text)
```

Before running it, decide whether the output will be `24`, `1212`, or an error. Python prints `1212`. The `+` operator joins two strings because both values are text. Quotation marks in source code are not decoration; they tell Python which kind of value to create.

Compare the integer version:

```python
quantity = 12
print(quantity + quantity)
```

This time `+` performs arithmetic and prints `24`. The visible digits are the same, but the **types** support different operations.

Run a wider comparison:

```python
whole = 12
measurement = 12.0
label = "12"
approved = True
missing = None

print(type(whole))
print(type(measurement))
print(type(label))
print(type(approved))
print(type(missing))
```

`type(value)` returns the value's type. Python displays `<class 'int'>`, `<class 'float'>`, `<class 'str'>`, `<class 'bool'>`, and `<class 'NoneType'>`. You do not need to memorize the angle-bracket display. The important names are `int`, `float`, `str`, `bool`, and `NoneType`.

A name refers to a value. The value—not the spelling of the name—has a type.



Ask for a compact type name

Types themselves have a `__name__` attribute:

```
PYTHON
temperature = 21.5
print(type(temperature).__name__)
```

Python first evaluates `type(temperature)`, which produces the `float` type. Then `.__name__` obtains the text name `"float"`. `print()` displays it without the `<class ...>` wrapper.

A value is not automatically converted to match an operation

Python refuses to guess how to combine unrelated types:

```
PYTHON
ticket_count = 4
message = "Tickets: "
print(message + ticket_count)
```

The last line raises `TypeError`. The message explains that Python can concatenate a string only with another string. Automatic guessing could conceal bad data, so the program must state the intended conversion.

2. Inspect a category with `isinstance()`

`type()` answers “what is the direct type of this value?” Another built-in, `isinstance()`, answers “does this value belong to this type or category?”

```
PYTHON
distance = 7.5

print(type(distance).__name__)
print(isinstance(distance, float))
print(isinstance(distance, str))
```

The results are `float`, `True`, and `False`. The second argument to `isinstance()` is a type, so it is written as `float` without quotes. The string `"float"` would merely be text, not the type itself.

Use `type()` while investigating an unexpected value. Use `isinstance()` when a program needs to ask whether a value belongs to an accepted type category.

A detail to recognize, not exploit

`bool` is a specialized kind of integer in Python, so `isinstance(True, int)` is `True` even though `type(True)` is `bool`. Write Boolean facts as `True` and `False`; do not use that relationship as a shortcut for arithmetic. Lesson 7 examines Boolean boundaries in detail. :::

Check the value you actually received

Predict all four results:

```python
participant_count = "8"

print(type(participant_count) is str)
print(isinstance(participant_count, str))
print(type(participant_count) is int)
print(isinstance(participant_count, int))
```

The first two are `True`; the last two are `False`. Both approaches agree for this ordinary string. The code does **not** prove that the text contains a valid participant count. Type inspection and content validation answer different questions.

Checkpoint: types and behavior

Types and behavior checkpoint

Predict Python’s result before choosing.

1. What does `"7" + "3"` produce?

- **A.** The string `"73"`
- **B.** The integer `10`
- **C.** The float `10.0`
- **D.** `None`

2. What does `type(4.5).__name__` produce?

- **A.** The number `4.5`
- **B.** The string `"float"`
- **C.** The string `"number"`
- **D.** `True`

3. What does `isinstance("eight", str)` prove?

- **A.** The text can be converted to `8`.
- **B.** The text is valid form input.
- **C.** The value is a string, but not what its characters mean.

- **D.** The value is an integer.

4. Why does `"Total: " + 4` fail?

- **A.** The integer is too small.
- **B.** `+` never works on text.
- **C.** The string is missing punctuation.
- **D.** Python will not guess whether to convert the string or the integer.

Write your choices before checking the answer key at the end of this book.

3. Conversion creates a result; it does not rewrite the source

Python's built-in type names can also act as conversion functions:

```python
quantity_text = "12"
quantity = int(quantity_text)

print(quantity_text, type(quantity_text).__name__)
print(quantity, type(quantity).__name__)
```

What Python does on each line:

1. `quantity_text = "12"` creates a string and binds the name.
2. `int(quantity_text)` reads the characters and creates the integer `12`.
3. Assignment binds the new integer to `quantity`.
4. Nothing reassigns `quantity_text`, so it still refers to the original string.

This distinction matters whenever the raw input is useful for error messages or auditing. You can preserve the source and give the converted value a new name.

Convert common numeric text

`int()` accepts optional surrounding whitespace and a sign:

```python
print(int("  42 "))
print(int("-7"))
print(int("+18"))
```

`float()` accepts decimal and scientific notation:

```python
print(float("3.75"))
print(float("-0.5"))
print(float("1.2e3"))
```

The final text means (1.2 \times 10^3), so Python prints `1200.0`.

Convert a value for display

`str()` creates text:

```python
attempts = 3
attempts_text = str(attempts)

print("Attempts: " + attempts_text)
print(type(attempts).__name__)
print(type(attempts_text).__name__)
```

The integer remains an integer. The new string is useful for text concatenation, although f-strings in Lesson 6 usually make mixed-value messages clearer.

Parse a non-decimal integer when the contract says so

The optional second argument to `int()` states the numeral base:

```python
binary_flags = int("1011", 2)
hex_color_part = int("ff", 16)

print(binary_flags)
print(hex_color_part)
```

Python prints `11` and `255`. Do this only when the input contract actually says binary or hexadecimal. `int("1011")` normally means the decimal number one thousand eleven.

4. A successful conversion can still lose information

Conversion does not promise that every detail survives:

```python
measurement = 8.9
whole_part = int(measurement)

print(whole_part)
print(measurement)
```

`int(8.9)` produces `8`. It truncates toward zero; it does not round to the nearest integer. The original `measurement` remains `8.9`, but `whole_part` no longer records the fractional part.

Compare positive and negative values:

```python
print(int(3.9))
print(int(-3.9))
```

The results are `3` and `-3`. Both move toward zero. If a ticket rule says “always use another seat for any partial group,” `int()` is the wrong operation. Lesson 2 will distinguish truncation, floor division, and rounding rules.

Numeric conversions can change the available precision

```python
large_count = 10_000_000_000_000_001
approximate_count = float(large_count)
restored_count = int(approximate_count)

print(large_count)
print(approximate_count)
print(restored_count)
print(restored_count == large_count)
```

On standard Python, converting this large integer to a float cannot preserve every digit. Converting it back does not recover the lost information. A round trip is not evidence of safety unless the representation can express the original value.

`bool()` answers truthiness, not human yes/no text

```python
print(bool("yes"))
print(bool("no"))
print(bool("False"))
print(bool(""))
```

The first three are `True` because every non-empty string is truthy. Only the empty string is false. Do not use `bool(user_text)` to parse a human answer. Lesson 7 builds an explicit yes/no rule.

Checkpoint: conversion boundaries

Conversion boundaries checkpoint

Choose the result that follows from the exact conversion.

1. After `number = int(text)` with `text = "15"`, what does `text` refer to?

- **A.** The original string `"15"`
- **B.** The integer `15`
- **C.** `None`
- **D.** A deleted value

2. What is `int(-4.8)`?

- **A.** `-5`
- **B.** `-4`
- **C.** `4`
- **D.** A `ValueError`

3. What is `bool("False")`?

- **A.** `False`, because the word says false
- **B.** `None`
- **C.** `True`, because the string is non-empty
- **D.** A `TypeError`

4. What does `int("101", 2)` produce?

- **A.** The decimal integer `101`
- **B.** The string `"5"`
- **C.** The float `5.0`
- **D.** The decimal integer `5`

Write your choices before checking the answer key at the end of this book.

5. Read conversion failures before repairing them

Try this valid Python code in its own cell:

```
PYTHON
participants = int("twelve")
```

Python raises `ValueError`. The call received the expected broad type—a string—but the characters are not a valid base-ten integer literal.

Now try:

```
PYTHON
participants = int(None)
```

This raises `TypeError`. `None` is not an accepted source for `int()`.

The distinction helps you describe the failure:

- `ValueError`**:** this kind of input is accepted, but this particular value cannot be interpreted as requested;
- `TypeError`**:** the operation does not accept this kind of value here.

Unit 8 will teach how to catch anticipated exceptions. For now, read the final line, identify the conversion call, and fix either the source data or the program's contract. Do not replace every failure with a made-up default.

Decimal-looking text is not an integer literal

```
PYTHON
whole = int("12.5")
```

This raises `ValueError`. If the contract permits decimal measurements, parse a float first:

```
PYTHON
measurement = float("12.5")
print(measurement)
```

Avoid writing `int(float("12.5"))` unless discarding the fractional part is an explicit rule. Making an error disappear is not the same as preserving meaning.

Test a conversion separately from a domain rule

```python
quantity_text = "-2"
quantity = int(quantity_text)

print(quantity)
print(quantity >= 0)
```

Conversion succeeds: `"-2"` is valid integer text. The comparison is `False` because a negative ticket quantity violates this example's domain rule. Keep the questions separate:

1. Can Python interpret the characters as an integer?
2. Does this program permit that integer?

6. `None` records deliberate absence

Suppose a registration form allows a nickname but does not require one:

```python
nickname = None
print(nickname is None)
```

`None` is a single special value used to represent "no value here" according to the program's contract. It is not the text `"None"`, not zero, and not an empty string.

Compare four possible states:

```python
unknown_score = None
zero_score = 0
blank_note = ""
literal_word = "None"

print(unknown_score is None)
print(zero_score is None)
print(blank_note is None)
print(literal_word is None)
```

Only the first result is `True`. Zero may be a real score. An empty string may mean the person submitted a blank note. The word `"None"` is ordinary text.

Check `None` with identity

Use:

```python
delivery_date = None
has_delivery_date = delivery_date is not None
print(has_delivery_date)
```

Python documentation and style guidance recommend `is None` and `is not None`. Lesson 6 of the course later examines equality and identity in depth. For now, treat this spelling as the standard `None` check.

Do not convert absence into misleading text

```python
special_request = None
request_text = str(special_request)
print(request_text)
print(type(request_text).__name__)
```

The output looks like `None`, but `request_text` is now a four-character string. It no longer represents absence. Convert to display text only at the display boundary, and choose a human message deliberately:

```python
special_request = None
display_request = "No special request"
print(display_request)
```

Later decision lessons will select that display based on the value. The key idea here is that representation carries meaning.

Checkpoint: missing, empty, and invalid

Missing and invalid values checkpoint

Choose the check or explanation that preserves the stated meaning.

1. Which expression checks whether `coupon_code` is deliberately missing?

- **A.** `coupon_code is None`
- **B.** `coupon_code == 0`
- **C.** `coupon_code == "None"`
- **D.** `int(coupon_code)`

2. Why should a score of `0` not automatically be replaced by `None`?

- **A.** Zero is always text.
- **B.** Zero can be a real measured score, while `None` means no score is present.
- **C.** `None` supports more arithmetic.
- **D.** Python converts zero to `None`.

3. Why does `int("twelve")` raise `ValueError`?

- **A.** `int` never accepts strings.
- **B.** The result would be too large.
- **C.** A string is accepted, but these characters are not an integer literal.
- **D.** The name `int` is missing.

4. `int("-2")` succeeds for a ticket quantity. What remains to check?

- **A.** Whether the string was deleted
- **B.** Whether `int` returned a float
- **C.** Whether quotes appear in output

- **D.** Whether the program permits a negative quantity

Write your choices before checking the answer key at the end of this book.

7. Build a typed registration summary

The observatory receives these fixed raw fields:

```
PYTHON
visitor_text = "  mina  "
group_size_text = "4"
donation_text = "12.50"
special_request_text = ""
```

Your task is to preserve the raw fields and create these new names:

- `visitor` — cleaned text with surrounding whitespace removed;
- `group_size` — an integer;
- `donation` — a float for this exercise;
- `special_request` — `None`, because the supplied text is empty; and
- `group_size_is_valid` — a Boolean showing whether the size is at least one.

You have not studied full decision structures yet, so use this starter for the optional field:

```
PYTHON
visitor = visitor_text.strip().title()
group_size = int(group_size_text)
donation = float(donation_text)
special_request = special_request_text.strip() or None
group_size_is_valid = group_size >= 1

print(f"Visitor: {visitor}")
print(f"Group size: {group_size}")
print(f"Donation: {donation}")
print(f"Request missing: {special_request is None}")
print(f"Group size valid: {group_size_is_valid}")
```

The expression `special_request_text.strip() or None` is a preview of Boolean operator behavior. Because the cleaned string is empty, the expression produces `None`. Lesson 7 will explain that rule; you only need to use the supplied line in this lab.

Verify each promise

Run these assertions after your code:

```
PYTHON
assert visitor_text == "  mina  "
assert visitor == "Mina"
assert type(group_size) is int
assert group_size == 4
assert type(donation) is float
assert donation == 12.5
assert special_request is None
assert group_size_is_valid is True
```

Then make one controlled variation at a time:

1 Change `group_size_text` to `"0"`. Which conversion still succeeds, and which domain fact changes?

2 Change `special_request_text` to `" wheelchair access "`. For this variation, assign `special_request = special_request_text.strip()` and verify the result.

3 Change `donation_text` to `"free"`. Record the exception type and the exact conversion that raises it.

4 Restore the ordinary data and rerun all assertions from a clean state.

Show the ordinary solution after completing your own version

PYTHON
```
visitor_text = "  mina  "
group_size_text = "4"
donation_text = "12.50"
special_request_text = ""

visitor = visitor_text.strip().title()
group_size = int(group_size_text)
donation = float(donation_text)
special_request = special_request_text.strip() or None
group_size_is_valid = group_size >= 1

print(f"Visitor: {visitor}")
print(f"Group size: {group_size}")
print(f"Donation: {donation}")
print(f"Request missing: {special_request is None}")
print(f"Group size valid: {group_size_is_valid}")
```

The raw names remain available. Each converted or cleaned value has a new name, so a reader can see where the program crosses from external text into values used by the program.

8. Check your typed boundary

Before leaving the lesson, explain these points without running the code again:

1 Why does `int(quantity_text)` not change `quantity_text`?

2 Which information does `int(8.9)` discard?

3 Why can `int("-2")` succeed even when `-2` is invalid for a particular form?

4 How do `None`, `0`, and `""` represent different states?

5 Which exception do you expect from `int("four")`, and what evidence supports that prediction?

If any answer feels uncertain, rerun the smallest relevant example rather than rereading the entire page.

Key points

Key points

- A Python value has a type that determines which operations it supports.
- `type()` identifies a direct type; `isinstance()` asks whether a value belongs to a type category.
- Conversion creates a result and does not rewrite the source value.
- Successful conversion can discard information, so the domain must choose the rule.
- `ValueError` often signals unacceptable contents; `TypeError` often signals an unsupported kind of value.

- `None` represents deliberate absence and is normally checked with `is None`.
- Conversion and domain validation answer different questions. :::

References

- Python built-in functions: `type`, `isinstance`, `int`, `float`, and `str`
- Python built-in constants: `None`, `True`, and `False`
- Python standard types: truth-value testing
- PEP 8 guidance for comparisons to `None`

Integers and Everyday Arithmetic

Online lesson and runnable notebook

Use Python integers to model exact counts, groups, leftovers, powers, ranges, and signed quantities without choosing the wrong arithmetic rule.

- **Level:** Python Foundations · Unit 2
- **Estimated time:** 2.5–4 hours
- **You will learn:** Choose integer operations for exact counts, complete groups, leftovers, powers, and signed values.
- **Practice in:** Google Colab, JupyterLab, or a local editor

1. Integers model exact whole quantities

An **integer** is a whole number with no fractional part. Counts, scores, row numbers, inventory changes, and elapsed whole seconds are natural integer values:

```python
visitors = 128
temperature_change = -3
empty_seats = 0

print(type(visitors).__name__)
print(type(temperature_change).__name__)
print(type(empty_seats).__name__)
```

All three values have type `int`. Zero and negative numbers are integers too. The minus sign is an operator applied to the literal `3`; the resulting value is negative three.

Digit separators improve large-number readability

Python lets source code contain underscores between digits:

```python
city_population = 1_250_000
same_population = 1250000

print(city_population)
print(city_population == same_population)
```

The underscore is for the person reading the code. It does not appear in the value or printed output, and the comparison is `True`.

Try grouping identifiers by meaning:

```python
binary_mask = 0b1111_0000
hex_color = 0xFF_88_00

print(binary_mask)
print(hex_color)
```

The `0b` and `0x` prefixes create binary and hexadecimal integer literals. They are still ordinary integers after Python reads them. Use these forms only when the domain naturally speaks in bits or hexadecimal; decimal remains clearer for ordinary counts.

Python integers are not limited to 32 bits

PYTHON
```
large_power = 2 ** 200
print(large_power)
print(type(large_power).__name__)
```

Python expands an integer to hold the result, limited by available memory rather than a fixed 32-bit or 64-bit boundary. That does not make huge calculations free, but it prevents silent wraparound in ordinary integer arithmetic.

2. Choose an operator from the question

Read this event-supply calculation from the inside out:

PYTHON
```
adult_tickets = 18
child_tickets = 7
ticket_price = 6

ticket_count = adult_tickets + child_tickets
ticket_income = ticket_count * ticket_price

print(ticket_count)
print(ticket_income)
```

`+` combines the two counts, producing `25`. `*` repeats the price for each ticket, producing `150`. Giving intermediate results names makes both questions visible.

The core integer operators are:

Operator	Question it can answer	Example	Result
`+`	What is the combined amount?	`8 + 3`	`11`
`-`	What is the difference or change?	`8 - 3`	`5`
`*`	How many across equal groups?	`8 * 3`	`24`
`/`	What is the exact ratio as a float?	`8 / 3`	`2.666...`
`//`	How many complete groups fit?	`8 // 3`	`2`
`%`	What remains after complete groups?	`8 % 3`	`2`
`**`	What is repeated multiplication?	`8 ** 3`	`512`

The table is a map, not a substitute for deciding what the program means.

Parentheses state the intended order

```python
weekday_visitors = 40
weekend_visitors = 65
weeks = 3

total_visitors = (weekday_visitors + weekend_visitors) * weeks
print(total_visitors)
```

The parentheses combine one weekday and weekend before multiplying by three. Without them, multiplication would happen before addition and answer a different question.

Useful built-ins keep intent visible

```python
altitude_change = -240
scores = (72, 91, 84)

print(abs(altitude_change))
print(min(72, 91, 84))
print(max(72, 91, 84))
print(pow(3, 4))
```

`abs()` gives distance from zero, `min()` and `max()` select boundary values, and `pow(3, 4)` is another spelling of `3 ** 4`. The tuple assigned to `scores` is a preview of Unit 3; the `min()` and `max()` calls above deliberately use separate arguments so no collection work is required.

Checkpoint: exact arithmetic

Exact arithmetic checkpoint

Choose the operation that answers the stated question.

1. Which expression calculates the total people in groups of 8 and 5?

- **A.** `8 + 5`
- **B.** `8 // 5`
- **C.** `8 % 5`
- **D.** `8 ** 5`

2. Which expression reports the size of a change stored as `-12`, ignoring direction?

- **A.** `min(-12)`
- **B.** `abs(-12)`
- **C.** `-12 % 2`
- **D.** `str(-12)`

3. Which expression doubles the combined values `a` and `b`?

- **A.** `a + b ** 2`
- **B.** `a + b * 2`

- **C.** `(a + b) * 2`
- **D.** `a ** b + 2`

4. What normally happens when a Python integer grows beyond 64 bits?

- **A.** It becomes a string.
- **B.** It silently becomes zero.
- **C.** It always becomes a float.
- **D.** Python allocates a larger integer, subject to available resources.

Write your choices before checking the answer key at the end of this book.

3. How many complete boxes can we fill?

Suppose `29` notebooks must be packed in boxes that hold `6` each:

```python
notebooks = 29
box_capacity = 6

complete_boxes = notebooks // box_capacity
leftover_notebooks = notebooks % box_capacity

print(complete_boxes)
print(leftover_notebooks)
```

Four complete boxes use `24` notebooks and leave `5`. Floor division `//` and remainder `%` describe two connected parts of the same grouping.

Verify the relationship:

```python
rebuilt_total = complete_boxes * box_capacity + leftover_notebooks
print(rebuilt_total)
print(rebuilt_total == notebooks)
```

For a positive capacity, the remainder is at least zero and smaller than the capacity. The equation

```text
total = quotient * divisor + remainder
```

is a useful way to check your model.

Get quotient and remainder together with `divmod()`

```python
complete_boxes, leftover_notebooks = divmod(29, 6)
print(complete_boxes)
print(leftover_notebooks)
```

`divmod(a, b)` returns both `a // b` and `a % b`. The two-name assignment is a small preview of tuple unpacking in Unit 3. Use the longer spelling if it is more readable at this point; the arithmetic is the same.

Convert seconds into minutes and seconds

```python
total_seconds = 367
minutes = total_seconds // 60
seconds = total_seconds % 60

print(minutes)
print(seconds)
```

Python prints `6` and `7`. Six complete minutes use `360` seconds, leaving seven. Change the total to `360`, `59`, and `0`. Predict each pair before running.

Count containers needed, including a partial one

Complete boxes and boxes needed are different questions. With 29 notebooks, five boxes are needed even though only four are full.

For positive integers:

```python
notebooks = 29
box_capacity = 6
boxes_needed = (notebooks + box_capacity - 1) // box_capacity
print(boxes_needed)
```

Adding `box_capacity - 1` moves any positive remainder into the next complete floor-division group. Test `30` notebooks: the formula still produces exactly five, not six.

Do not reuse the ceiling formula blindly

`(amount + size - 1) // size` assumes positive integer amounts and sizes. A negative amount or zero size needs a different contract. Later lessons will show `math.ceil()` for general numeric values. :::

4. `/` and `//` answer different questions

Even when both operands are integers, `/` produces a float:

```python
print(12 / 3)
print(type(12 / 3).__name__)
print(12 // 3)
print(type(12 // 3).__name__)
```

The outputs are `4.0`, `float`, `4`, and `int`. Use `/` for a ratio or average and `//` for complete floor-division groups.

Average is not a complete-group count

```python
total_points = 25
players = 4

average_points = total_points / players
complete_points_each = total_points // players

print(average_points)
print(complete_points_each)
```

The average is `6.25`. If points must be indivisible objects distributed equally, each player can receive six complete points with one left over. The correct operator comes from the requirement.

Division by zero has no numeric result

Run this in its own cell:

```python
groups = 0
items_per_group = 12 // groups
```

Python raises `ZeroDivisionError`. Changing `groups` to `1` removes the exception, but a real program must ask why zero reached a value used as a divisor. Unit 8 will teach controlled handling; for now, preserve the message and repair the source or stated rule.

The remainder operator also rejects a zero divisor:

```python
remainder = 12 % 0
```

The failure belongs to the operation, not to the integer `12` by itself.

Checkpoint: groups, ratios, and leftovers

Groups and leftovers checkpoint

Translate the question into the matching integer operation.

1. How many complete groups of 6 fit in 29?

- **A.** `29 // 6`
- **B.** `29 / 6`
- **C.** `29 % 6`
- **D.** `6 ** 29`

2. Which expression finds the notebooks left after packing complete boxes of 6?

- **A.** `notebooks / 6`
- **B.** `notebooks % 6`
- **C.** `notebooks ** 6`
- **D.** `6 - notebooks`

3. Which expression calculates a possibly fractional average for `total` across `people`?

- **A.** `total % people`
- **B.** `total // people`
- **C.** `total / people`
- **D.** `people / total // 1`

4. What two values does `divmod(367, 60)` return?

- **A.** `367` and `60`

- **B.** `6.116...` and `0`
- **C.** `7` and `6`
- **D.** `6` and `7`

Write your choices before checking the answer key at the end of this book.

5. Negative floor division follows the lower integer

For positive values, floor division often looks like discarding the fractional part. Negative values reveal the real rule:

```
PYTHON
print(7 / 3)
print(7 // 3)
print(-7 / 3)
print(-7 // 3)
```

`-7 / 3` is about `-2.333`. The mathematical floor is the next lower integer, `-3`, so `-7 // 3` is `-3`. It does not truncate toward zero.

Compare `int()` from Lesson 1:

```
PYTHON
print(int(-7 / 3))
print(-7 // 3)
```

`int(-2.333...)` truncates toward zero and gives `-2`. Floor division gives `-3`. The operations implement different rules.

Remainder keeps the grouping equation true

```
PYTHON
quotient = -7 // 3
remainder = -7 % 3

print(quotient)
print(remainder)
print(quotient * 3 + remainder)
```

Python prints `-3`, `2`, and `-7`. The remainder has the same sign as the positive divisor and keeps the reconstruction equation true.

A clock-style use makes this behavior helpful:

```
PYTHON
hour = 1
hours_back = 5
wrapped_hour = (hour - hours_back) % 24
print(wrapped_hour)
```

The result is `20`, representing 20:00 on a 24-hour clock. Modulo wraps a negative intermediate value into the range from `0` through `23`.

6. Powers grow faster than repeated addition

Exponentiation models repeated multiplication:

```python
square_side = 12
square_area = square_side ** 2
cube_volume = 3 ** 3

print(square_area)
print(cube_volume)
```

`12 ** 2` means twelve squared, while `3 ** 3` means three multiplied by itself three times.

Operator precedence matters around a negative sign:

```python
print(-3 ** 2)
print((-3) ** 2)
```

Exponentiation happens before the leading negative operator, so `-3 ** 2` is `-(3 ** 2)`, or `-9`. Parentheses make the negative value itself the base, giving `9`.

Checkpoint: signed arithmetic

Signed arithmetic checkpoint

Trace the exact rule rather than relying on a visual shortcut.

1. What is `-7 // 3`?

- **A.** `-3`
- **B.** `-2`
- **C.** `2`
- **D.** `3`

2. What is `-7 % 3` in Python?

- **A.** `-1`
- **B.** `2`
- **C.** `-2`
- **D.** `1`

3. What is `(-3) ** 2`?

- **A.** `-9`
- **B.** `-6`
- **C.** `9`
- **D.** `6`

4. Why can `int(-2.8)` differ from `-28 // 10`?

- **A.** Only integers can be negative.
- **B.** `//` converts text.
- **C.** `int` always rounds to the nearest integer.
- **D.** `int` truncates toward zero, while `//` floors toward the lower integer.

Write your choices before checking the answer key at the end of this book.

7. Build a festival crate planner

A community festival has `367` water bottles. A crate holds `24` bottles, and a volunteer cart can carry `5` crates per trip. Build a summary that reports:

- full crates;
- bottles left outside full crates;
- total crates needed to carry every bottle;
- complete cart trips;
- crates on the final partial trip; and
- total cart trips needed.

Start with these names:

```python
bottles = 367
crate_capacity = 24
crates_per_trip = 5

full_crates = None
leftover_bottles = None
crates_needed = None
complete_trips = None
crates_on_partial_trip = None
trips_needed = None
```

Replace each `None` with an integer expression. Keep the two questions separate: `full_crates` counts only completely filled crates, while `crates_needed` includes the partially filled final crate.

Use these checks:

```python
assert full_crates == 15
assert leftover_bottles == 7
assert crates_needed == 16
assert complete_trips == 3
assert crates_on_partial_trip == 1
assert trips_needed == 4
assert full_crates * crate_capacity + leftover_bottles == bottles
```

Then format a short report:

```python
print(f"Bottles: {bottles}")
print(f"Crates needed: {crates_needed}")
print(f"Cart trips needed: {trips_needed}")
```

Test the exact-boundary case

Change `bottles` to `360`. Recalculate every derived value rather than editing the assertions until they pass. Predict which remainders become zero and whether the crate/trip totals still need a partial group.

Hint 1: separate full groups from groups needed

Use `//` and `%` for the full-crate facts. Use the positive-integer ceiling formula for crates and trips needed.

Show one solution after your assertions pass

```python
bottles = 367
crate_capacity = 24
crates_per_trip = 5

full_crates = bottles // crate_capacity
leftover_bottles = bottles % crate_capacity
crates_needed = (bottles + crate_capacity - 1) // crate_capacity
complete_trips = crates_needed // crates_per_trip
crates_on_partial_trip = crates_needed % crates_per_trip
trips_needed = (crates_needed + crates_per_trip - 1) // crates_per_trip
```

`complete_trips` is `3` because three carts can be completely filled with five crates. `trips_needed` is `4` because the remaining crate still needs transport.

8. Check your arithmetic model

Explain these choices in your own words:

1. Why does `/` produce the wrong type of answer for a complete-box count?
2. How do `//` and `%` describe the same grouping?
3. Why does `int(-2.8)` not implement mathematical floor?
4. Which assumption makes the positive-integer ceiling formula safe?
5. How would you detect a zero divisor before attempting division?

Key points

Key points

- Integers model exact whole quantities and can grow beyond fixed machine-word sizes.
- Choose `+`, `-`, `*`, `/`, `//`, `%`, and `**` from the question being answered.
- `/` returns a floating-point ratio; `//` returns the floor-division result.
- Quotient and remainder satisfy `total == quotient * divisor + remainder`.
- Negative floor division moves toward the lower integer, not toward zero.
- A complete-group count and the number of containers needed are different facts.
- Zero cannot be used as a divisor. :::

References

- Python tutorial: using Python as a calculator
- Python standard types: numeric operations
- Python reference: arithmetic conversions and operations
- Python built-in functions: `abs`, `divmod`, `min`, `max`, and `pow`

Floating-Point Numbers and Precision

Online lesson and runnable notebook

Use floats for measured quantities, compare approximate results with suitable tolerances, and separate rounded displays from stored values.

- **Level:** Python Foundations · Unit 2
- **Estimated time:** 3–4 hours
- **You will learn:** Model measured quantities with floats, explain binary approximation, compare with tolerances, and format results without mistaking display rounding for exactness.
- **Practice in:** Google Colab, JupyterLab, or a local editor

1. Floats model quantities between whole numbers

A trail can be `4.75` kilometers long, a sensor can report `21.6` degrees, and an experiment can take `0.003` seconds. Python normally represents these measured or fractional values with `float`:

```python
trail_km = 4.75
temperature_c = 21.6
elapsed_seconds = 0.003

print(type(trail_km).__name__)
print(type(temperature_c).__name__)
print(type(elapsed_seconds).__name__)
```

The name “floating point” refers to a representation whose decimal point can effectively move as the magnitude changes. It covers very small and very large values with a finite amount of storage.

Scientific notation keeps scale readable

```python
speed_of_light = 2.99792458e8
proton_width_m = 8.4e-16

print(speed_of_light)
print(proton_width_m)
```

`2.99792458e8` means (2.99792458 \times 10^8). `8.4e-16` means (8.4 \times 10^{-16}). These are float literals, not strings.

Integer and float arithmetic can mix

```python
distance_km = 15
hours = 2.5
average_speed = distance_km / hours

print(average_speed)
print(type(average_speed).__name__)
```

Python converts the integer to a compatible numeric representation for the operation. The result is the float `6.0`. The original name `distance_km` still refers to an integer; the conversion for the calculation does not reassign it.

A float carries limited precision

PYTHON
```
print(1.5 + 2.25)
print(9.0 / 4.0)
print(2.0 ** -3)
```

These results are `3.75`, `2.25`, and `0.125`. Fractions whose denominator is a power of two can often be represented exactly in binary. Many familiar decimal fractions cannot.

2. Why `0.1 + 0.2` is not exactly `0.3`

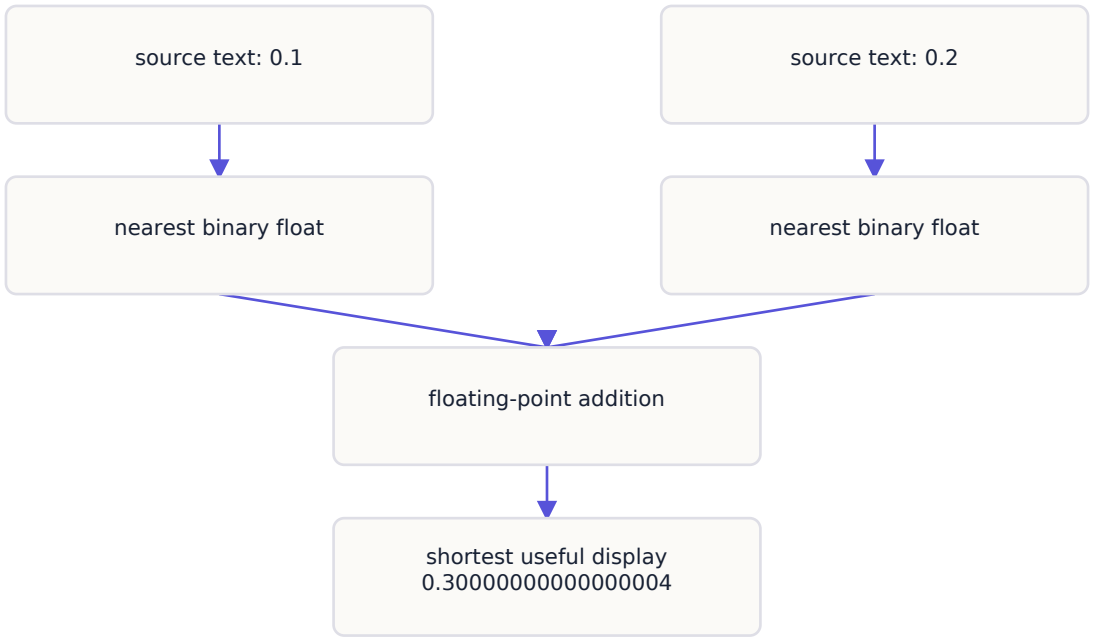
Run the classic comparison:

PYTHON
```
result = 0.1 + 0.2

print(result)
print(result == 0.3)
```

Python commonly prints `0.30000000000000004`, and the equality comparison is `False`. This is not random. A float stores a number in binary using a finite number of bits. Just as decimal `1 / 3` becomes the repeating `0.333333...`, decimal `0.1` repeats in binary.

The written decimal is converted to the nearest available binary float. Arithmetic uses those stored approximations.



Python’s normal display chooses the shortest decimal text that identifies the stored float. It does not display every internal binary digit, but it sometimes shows enough digits to reveal the approximation.

Inspect more digits with formatting

```python
value = 0.1
print(f"{value:.20f}")
```

The output is close to `0.10000000000000000555`. The final digits make clear that the stored value is near one tenth, not the exact decimal fraction one tenth.

Compare a binary-friendly value:

```python
value = 0.125
print(f"{value:.20f}")
```

One eighth is exactly representable in binary, so the displayed digits after `125` are zeros.

Approximation is usually appropriate for measurements

A ruler reading such as `12.4 cm` is already limited by the instrument and human reading. A nearby binary representation is normally much more precise than the measurement itself. The mistake is not using floats; the mistake is demanding exact decimal behavior when the problem requires it.

Checkpoint: float representation

Float representation checkpoint

Choose the explanation that matches Python’s stored values.

1. Which value is a natural use of `float`?

- **A.** A measured temperature of 21.6 °C
- **B.** An exact count of 12 seats
- **C.** A missing delivery date
- **D.** A person’s name

2. Why can `0.1 + 0.2 == 0.3` be `False`?

- **A.** Python sometimes skips addition.
- **B.** Those decimal fractions are stored as nearby finite binary values.
- **C.** Equality cannot compare floats.
- **D.** The result is converted to text first.

3. Which decimal fraction is exactly representable as a finite binary fraction?

- **A.** `0.1`
- **B.** `0.2`
- **C.** `0.125`
- **D.** `0.3`

4. What does `1.2e3` represent?

- **A.** `1.23`

- **B.** `123`
- **C.** The text `"1.2e3"`
- **D.** The float `1200.0`

Write your choices before checking the answer key at the end of this book.

3. Compare measurements with a stated tolerance

Exact equality asks whether two floats are the same stored number:

```python
calculated = 0.1 + 0.2
expected = 0.3

print(calculated == expected)
print(abs(calculated - expected))
```

The difference is extremely small but nonzero. For approximate quantities, ask whether the difference is acceptable for the task.

Python's `math.isclose()` implements a clear tolerance comparison:

```python
import math

calculated = 0.1 + 0.2
expected = 0.3

print(math.isclose(calculated, expected))
```

The default tolerances consider these values close, so the output is `True`. Default settings are convenient for demonstrations, but real tolerances should come from a requirement such as instrument accuracy.

Relative tolerance scales with magnitude

```python
import math

expected_distance = 1_000.0
measured_distance = 1_000.5

print(math.isclose(measured_distance, expected_distance, rel_tol=0.001))
print(math.isclose(measured_distance, expected_distance, rel_tol=0.0001))
```

A relative tolerance of `0.001` allows roughly one part in a thousand, which is about `1.0` at this scale. A tolerance of `0.0001` allows roughly `0.1`, so the half-unit difference is too large.

Absolute tolerance matters near zero

Relative comparison alone is not useful when the expected value is zero:

```python
import math

sensor_offset = 0.0004

print(math.isclose(sensor_offset, 0.0))
```

PYTHON (CONTINUED)

```
print(math.isclose(sensor_offset, 0.0, abs_tol=0.001))
```

The second comparison states that offsets within `0.001` of zero are acceptable. Choose the unit and tolerance together: `0.001` meters is not the same requirement as `0.001` kilometers.

Tolerance is a domain rule

Do not add an enormous tolerance merely to make a test pass. State what difference the instrument, user, or calculation is allowed to have, then encode that rule. :::

Exact equality is still useful

Float equality is appropriate when testing an exact sentinel that your own code assigned, or when the contract truly requires the same stored value. The rule is not “never use `==` with floats.” It is “do not use exact equality to express an approximate requirement.”

PYTHON

```
reading = 0.0
not_started = reading == 0.0
print(not_started)
```

If `reading` came from a physical sensor, an absolute tolerance might be the better meaning. Context decides.

4. Rounding and formatting do different jobs

`round()` creates a numeric result:

PYTHON

```
measurement = 12.34567
rounded_measurement = round(measurement, 2)

print(rounded_measurement)
print(type(rounded_measurement).__name__)
print(measurement)
```

The result is a float near `12.35`. The original value remains unchanged.

Formatting creates text for display:

PYTHON

```
measurement = 12.34567
display_text = f"{measurement:.2f}"

print(display_text)
print(type(display_text).__name__)
print(measurement)
```

`display_text` is the string `"12.35"`. The stored measurement is still the original float. Use formatting when the need is “show two decimal places.”

Display precision does not change later arithmetic

PYTHON

```
distance = 2.675
print(f"Displayed: {distance:.2f}")
print(distance * 10)
```

The formatted output does not feed back into `distance`. Later arithmetic uses the stored float, not the characters shown to a person.

`round()` follows the stored value and a tie rule

```python
print(round(2.5))
print(round(3.5))
```

Python normally uses “round half to even” for exact halfway cases: the results are `2` and `4`. With values such as `2.675`, binary approximation means the stored value may not lie on the decimal halfway point you imagine.

```python
print(round(2.675, 2))
```

Do not build legal or financial rules from an assumed school-rounding behavior. The next lesson uses `Decimal` with an explicit rounding mode.

Checkpoint: comparisons and display

Float comparison and display checkpoint

Match the operation to the promise the program needs to make.

1. Which check best expresses that a measured value may differ from zero by at most 0.01 units?

- **A.** `math.isclose(value, 0.0, abs_tol=0.01)`
- **B.** `value == 0.01`
- **C.** `round(value) == value`
- **D.** `str(value) == "0"`

2. What type does `f"{3.14159:.2f}"` produce?

- **A.** `float`
- **B.** `str`
- **C.** `int`
- **D.** `bool`

3. After `shown = f"{value:.1f}"`, which value is used by `value * 2`?

- **A.** The characters in `shown`
- **B.** A value rounded to an integer
- **C.** The original float bound to `value`
- **D.** `None`

4. Who should determine whether a difference of 0.5 is acceptable?

- **A.** The shortest code
- **B.** The `print()` function

- **C.** The float type alone
- **D.** The measurement or application requirement

Write your choices before checking the answer key at the end of this book.

5. Repeated operations can accumulate error

Adding one tenth ten times illustrates accumulation:

```python
total = 0.0
total = total + 0.1
total = total + 0.1
total = total + 0.1
total = total + 0.1
total = total + 0.1
total = total + 0.1
total = total + 0.1
total = total + 0.1
total = total + 0.1
total = total + 0.1

print(total)
print(total == 1.0)
```

This intentionally repeats statements because loops are taught later. The stored result may be `0.9999999999999999`. Compare it according to the requirement:

```python
import math

print(math.isclose(total, 1.0))
```

For long sums, Python's `math.fsum()` can reduce accumulated rounding error. It accepts a collection, so this example previews a list without requiring you to modify it:

```python
import math

values = [0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1]
print(sum(values))
print(math.fsum(values))
```

Unit 3 will teach lists and iteration. For now, observe that algorithm choice can affect accumulated error even when every input is the same.

6. Recognize infinity and NaN at numeric boundaries

Some calculations or external numeric systems produce special floats:

```python
positive_infinity = float("inf")
not_a_number = float("nan")

print(positive_infinity)
print(not_a_number)
```

Infinity is larger than ordinary finite floats:

```python
print(float("inf") > 1e308)
```

NaN means “not a number” and deliberately does not compare equal to itself:

```python
not_a_number = float("nan")
print(not_a_number == not_a_number)
```

The result is `False`. Do not use equality to detect NaN. The `math` module has explicit checks:

```python
import math

print(math.isfinite(12.5))
print(math.isfinite(float("inf")))
print(math.isnan(float("nan")))
```

Ordinary division by zero in Python still raises `ZeroDivisionError`; it does not automatically produce infinity:

```python
result = 1.0 / 0.0
```

Special values often arrive from scientific libraries or external data. Recognize them now; later data and scientific courses will establish policies for them.

Checkpoint: numeric boundaries

Floating-point boundaries checkpoint

Choose the result or check that Python actually provides.

1. Which expression is true only for an ordinary finite float?

- **A.** `math.isfinite(value)`
- **B.** `value == float("nan")`
- **C.** `str(value) != value`
- **D.** `round(value) is value`

2. What is the reliable way to detect a NaN value?

- **A.** Compare it with itself.
- **B.** Use `math.isnan(value)`.
- **C.** Round it to zero.
- **D.** Convert it to an integer.

3. What does ordinary Python evaluate for `1.0 / 0.0`?

- **A.** Positive infinity
- **B.** NaN
- **C.** It raises `ZeroDivisionError`

- **D.** The integer zero

4. Why can repeatedly adding `0.1` finish slightly away from the expected decimal total?

- **A.** Assignment sometimes skips a line.
- **B.** Addition changes the source code.
- **C.** Python converts every sum to an integer.
- **D.** Each operation works with finite binary approximations whose errors can accumulate.

Write your choices before checking the answer key at the end of this book.

7. Build a sensor calibration report

An observatory expects a calibration weight of `10.0` grams. Three repeated readings are `10.004`, `9.998`, and `10.002` grams. For this instrument, a mean within `0.005` grams of the target is acceptable.

Use the supplied names without a loop:

```python
target_grams = 10.0
reading_1 = 10.004
reading_2 = 9.998
reading_3 = 10.002
tolerance_grams = 0.005

mean_grams = None
difference_grams = None
calibrated = None
```

Calculate the mean, absolute difference, and calibration fact. Then display the mean and difference to three decimal places.

Use these assertions:

```python
import math

assert math.isclose(mean_grams, 10.001333333333333)
assert math.isclose(difference_grams, 0.0013333333333331865)
assert calibrated is True
assert f"{mean_grams:.3f}" == "10.001"
assert f"{difference_grams:.3f}" == "0.001"
```

Keep values and displays separate

Calculate `calibrated` from the unformatted mean and the stated tolerance. Do not compare display strings or round the measurements first. :::

After the ordinary case passes, change `reading_3` to `10.030`. Predict whether the mean stays within tolerance, then recalculate from the top.

Hint: translate the requirement directly

The mean is the sum divided by `3`. The difference is `abs(mean_grams - target_grams)`. `math.isclose()` can compare the two values with `abs_tol=tolerance_grams` and `rel_tol=0.0` because the requirement is stated as an absolute difference in grams.

Show one solution after testing your report

```python
import math

target_grams = 10.0
reading_1 = 10.004
reading_2 = 9.998
reading_3 = 10.002
tolerance_grams = 0.005

mean_grams = (reading_1 + reading_2 + reading_3) / 3
difference_grams = abs(mean_grams - target_grams)
calibrated = math.isclose(
    mean_grams,
    target_grams,
    rel_tol=0.0,
    abs_tol=tolerance_grams,
)

print(f"Mean: {mean_grams:.3f} g")
print(f"Difference: {difference_grams:.3f} g")
print(f"Calibrated: {calibrated}")
```

The multiline call is one expression inside open parentheses. The four-space continuation indentation makes its arguments easy to inspect.

8. Check your precision decisions

Before moving on, answer these questions from the sensor artifact:

1. Why is float appropriate for measured grams but not automatically for exact decimal currency?
2. What tolerance did the requirement state, and in which unit?
3. Why should formatting happen after the calibration comparison?
4. How would the contract change if the tolerance were a percentage of the target?
5. Which explicit function detects a non-finite reading?

Key points

Key points

- Floats are finite binary approximations suitable for many measured quantities.
- Familiar decimal fractions such as `0.1` may not have exact finite binary forms.
- `math.isclose()` expresses approximate equality with relative and absolute tolerances.
- A tolerance belongs to the problem and must use the value's unit.
- `round()` produces a numeric result; formatting produces display text.
- Rounded output does not change the stored float used by later arithmetic.
- `math.isfinite()` and `math.isnan()` recognize special floating-point values. :::

References

- Python tutorial: floating-point arithmetic and its limitations
- Python `math.isclose()` documentation

- Python `math` checks for finite, infinite, and NaN values
- Python format-specification mini-language

Money, Exact Ratios, and Complex Numbers

Online lesson and runnable notebook

Choose Decimal, Fraction, or complex when a problem needs exact base-ten rules, exact ratios, or real and imaginary components.

- **Level:** Python Foundations · Unit 2
- **Estimated time:** 3–4.5 hours
- **You will learn:** Choose and use `Decimal`, `Fraction`, and `complex` values according to the numeric rules of a task.
- **Practice in:** Google Colab, JupyterLab, or a local editor

1. Choose a number type from the rule you must preserve

Python has more than one useful numeric representation because different problems make different promises:

Requirement	Useful representation	Example
Exact whole count	`int`	27 participants
Measured approximation	`float`	21.6 °C
Base-ten arithmetic and explicit decimal rounding	`Decimal`	19.95 credits
Exact ratio of integers	`Fraction`	2/3 of a recipe
Real and imaginary components	`complex`	3 + 4j

No type is “the most accurate” for every job. Accuracy means preserving the rule the program promises.

Start by comparing three representations of one tenth:

```python
from decimal import Decimal
from fractions import Fraction

binary_approximation = 0.1
decimal_tenth = Decimal("0.1")
fraction_tenth = Fraction(1, 10)

print(binary_approximation)
print(decimal_tenth)
print(fraction_tenth)
```

The output looks like `0.1`, `0.1`, and `1/10`, but the stored rules differ. The float is the nearest available binary value. The Decimal stores the supplied base-ten digits. The Fraction stores an exact numerator and denominator.

Choose once at the boundary

Read the input contract, choose the representation, and keep arithmetic in that representation. Casual mixing can produce errors or quietly carry an earlier approximation into a supposedly exact calculation. :::

2. `Decimal` follows base-ten arithmetic

`Decimal` lives in Python's standard-library `decimal` module:

```python
from decimal import Decimal

unit_price = Decimal("3.25")
quantity = 4
subtotal = unit_price * quantity

print(subtotal)
print(type(subtotal).__name__)
```

The result is `13.00`. Decimal preserves the two written fractional digits during this multiplication. It can multiply by an integer without changing numeric systems.

Construct decimals from strings

Compare two constructors:

```python
from decimal import Decimal

from_text = Decimal("0.1")
from_float = Decimal(0.1)

print(from_text)
print(from_float)
print(from_text == from_float)
```

The float already contains a binary approximation. `Decimal(0.1)` faithfully converts that approximation, exposing many digits. It cannot reconstruct the exact decimal characters that were lost earlier. When external data states base-ten digits, construct from the original string.

Ordinary Decimal arithmetic

```python
from decimal import Decimal

price = Decimal("7.40")
discount = Decimal("1.25")
quantity = 3

subtotal = price * quantity
discounted_total = subtotal - discount
share = discounted_total / Decimal("2")

print(subtotal)
print(discounted_total)
print(share)
```

Every non-integer numeric operand is a Decimal. This makes the arithmetic rule visible to a reader.

Quantize to a declared money unit

Many currencies use a smallest displayed/accounted unit such as `0.01`. The program must also choose how halfway values round:

```python
from decimal import Decimal, ROUND_HALF_UP

raw_total = Decimal("19.995")
cent = Decimal("0.01")
money_total = raw_total.quantize(cent, rounding=ROUND_HALF_UP)

print(money_total)
```

The result is `20.00`. `quantize()` shapes the result to the exponent of `0.01`, and `ROUND_HALF_UP` states the tie rule. Do not assume this rule is correct for every currency, tax authority, or organization; make the domain policy explicit.

Round at the right stage

```python
from decimal import Decimal, ROUND_HALF_UP

unit_price = Decimal("1.235")
quantity = 3
cent = Decimal("0.01")

round_each_first = unit_price.quantize(cent, rounding=ROUND_HALF_UP) * quantity
round_total_last = (unit_price * quantity).quantize(
    cent,
    rounding=ROUND_HALF_UP,
)

print(round_each_first)
print(round_total_last)
```

The results differ: `3.72` versus `3.71`. Both calculations run correctly; only the invoice rule can say which stage is correct. Record whether the contract rounds each line item, each unit, tax, or only the final total.

Decimal precision is a context

Decimal arithmetic has a current precision setting:

```python
from decimal import Decimal, localcontext

one = Decimal("1")
seven = Decimal("7")

with localcontext() as context:
    context.prec = 8
    print(one / seven)

print(one / seven)
```

The `with` block temporarily uses eight significant digits. The surrounding context is restored afterward. `with` will be taught in depth with resources; here it provides a safe local demonstration rather than changing the entire notebook’s Decimal settings.

Do not mix Decimal and float casually

Run this in its own cell:

```python
from decimal import Decimal

total = Decimal("2.50") + 0.5
```

Python raises `TypeError` for Decimal-plus-float arithmetic. Decide whether the source is decimal text or an approximate measurement, then convert at the boundary. The error prevents an ambiguous promise.

Checkpoint: decimal rules

Decimal arithmetic checkpoint

Choose the representation or operation that preserves the stated base-ten rule.

1. A form sends the exact price text `"0.10"`. Which constructor preserves those stated digits?

- **A.** `Decimal("0.10")`
- **B.** `Decimal(0.10)`
- **C.** `float(Decimal("0.10"))`
- **D.** `int("0.10")`

2. What does `value.quantize(Decimal("0.01"), rounding=ROUND_HALF_UP)` make explicit?

- **A.** A binary bit width
- **B.** A two-decimal quantum and a tie-breaking rule
- **C.** A currency symbol
- **D.** An integer-only result

3. Who decides whether each item or only the final invoice total should be rounded?

- **A.** The shortest expression
- **B.** The `Decimal` class name
- **C.** The invoice or domain contract
- **D.** The number of print statements

4. Why does Python reject `Decimal("1.0") + 0.5`?

- **A.** Decimal cannot add.
- **B.** The values have different signs.
- **C.** The result would always be zero.
- **D.** The expression mixes decimal and binary-float arithmetic without an explicit conversion decision.

Write your choices before checking the answer key at the end of this book.

3. `Fraction` keeps ratios exact

`Fraction` stores an integer numerator over an integer denominator:

```python
from fractions import Fraction

recipe_share = Fraction(2, 3)

print(recipe_share)
print(recipe_share.numerator)
print(recipe_share.denominator)
```

Python displays `2/3`. The attributes expose the normalized numerator and denominator.

Fractions reduce automatically

```python
from fractions import Fraction

half = Fraction(6, 12)
negative_half = Fraction(3, -6)

print(half)
print(negative_half)
```

The results are `1/2` and `-1/2`. Python reduces common factors and places the sign in the numerator.

Arithmetic remains exact

```python
from fractions import Fraction

morning = Fraction(1, 3)
afternoon = Fraction(1, 6)
combined = morning + afternoon
remaining = 1 - combined

print(combined)
print(remaining)
```

Both results are exactly `1/2`. The integer `1` can participate without losing the fraction representation.

Use multiplication and division for scaling:

```python
from fractions import Fraction

original_cups = Fraction(3, 4)
scale = Fraction(5, 2)
scaled_cups = original_cups * scale

print(scaled_cups)
print(float(scaled_cups))
```

The exact result is `15/8`. Conversion to float produces `1.875` for a measuring display. Keep the Fraction for later exact calculations.

Build from text or Decimal when possible

```python
from fractions import Fraction

print(Fraction("0.125"))
print(Fraction("3/8"))
```

These produce exact `1/8` and `3/8`. Constructing from a float preserves the exact binary float ratio, which may be surprising:

```python
from fractions import Fraction

print(Fraction(0.1))
```

The numerator and denominator are large because the input was already a binary approximation.

Recover a simple nearby ratio deliberately

```python
from fractions import Fraction

approximation = Fraction(0.3333333333333333)
simple_ratio = approximation.limit_denominator(100)

print(approximation)
print(simple_ratio)
```

`limit_denominator(100)` asks for the closest fraction whose denominator is at most 100, producing `1/3`. That limit is a policy: a stricter or larger limit can produce a different rational approximation.

Zero cannot be a denominator

```python
from fractions import Fraction

invalid_ratio = Fraction(3, 0)
```

Python raises `ZeroDivisionError`. A Fraction represents a numeric ratio, and division by zero has no finite rational value.

Checkpoint: exact ratios

Exact ratios checkpoint

Trace how Fraction preserves and normalizes a ratio.

1. What does `Fraction(6, 12)` display?

- **A.** `1/2`
- **B.** `6/12` only, with no normalization
- **C.** `0`
- **D.** `12/6`

2. What is `Fraction(1, 3) + Fraction(1, 6)`?

- **A.** `2/9`
- **B.** `1/2`
- **C.** Approximately `0.4` only
- **D.** `2/6` without reduction

3. Why can `Fraction(0.1)` have a large numerator and denominator?

- **A.** Fraction chooses random digits.
- **B.** The denominator is always a power of ten.
- **C.** It exactly represents the already-approximate binary float.
- **D.** Fractions cannot represent tenths.

4. What rule does `value.limit_denominator(100)` add?

- **A.** The numerator must be zero.
- **B.** The result must be an integer.
- **C.** The value must become a Decimal.
- **D.** Choose a nearby fraction whose denominator is at most 100.

Write your choices before checking the answer key at the end of this book.

4. `complex` combines real and imaginary components

Complex numbers are common in electrical engineering, waves, rotations, and scientific models. Python writes the imaginary component with `j`:

```python
position = 3 + 4j

print(position)
print(type(position).__name__)
print(position.real)
print(position.imag)
```

The real component is `3.0` and the imaginary component is `4.0`. These attributes are floats even when the literal components look like integers.

Magnitude is distance from the origin

```python
position = 3 + 4j
distance = abs(position)
print(distance)
```

The magnitude is `5.0`, following the right-triangle relationship between the real and imaginary components.

Arithmetic follows complex-number rules

```python
signal_a = 2 + 3j
signal_b = 1 - 1j

print(signal_a + signal_b)
print(signal_a * signal_b)
```

Addition combines corresponding components. Multiplication uses `j * j == -1`. Predict the components before running, then calculate them by hand.

Conjugates reverse the imaginary sign

```python
signal = 2 + 3j
conjugate = signal.conjugate()

print(conjugate)
print(signal * conjugate)
```

The conjugate is `2 - 3j`. Multiplying a complex value by its conjugate produces the real value corresponding to magnitude squared—in this case `(13+0j)`.

Ordering complex numbers is not defined

```python
print((2 + 1j) < (3 + 0j))
```

Python raises `TypeError`. Complex values do not have one natural less-than order. Compare a meaningful derived quantity such as magnitude if the domain defines that rule:

```python
first = 2 + 1j
second = 3 + 0j
print(abs(first) < abs(second))
```

The comparison now has an explicit meaning: which value is farther from the origin.

Checkpoint: complex components

Complex-number checkpoint

Choose the result that follows from real and imaginary components.

1. Which Python literal has real part 3 and imaginary part 4?

- **A.** `3 + 4j`
- **B.** `3 + 4i`
- **C.** `complex[3, 4]`
- **D.** `"3 + 4j"`

2. What is `abs(3 + 4j)`?

- **A.** `7.0`
- **B.** `5.0`

- **C.** `1.0`
- **D.** `12.0`

3. What is `(2 + 3j).conjugate()`?

- **A.** `-2 - 3j`
- **B.** `3 + 2j`
- **C.** `2 - 3j`
- **D.** `2 + 3j`

4. How should a program compare two complex values when it needs "closer to the origin"?

- **A.** Use `<` directly on the complex values.
- **B.** Convert both to strings.
- **C.** Compare only their real parts without saying so.
- **D.** Compare their magnitudes with `abs()`.

Write your choices before checking the answer key at the end of this book.

5. Complete a three-station number lab

This lab asks you to make three representation decisions. Keep each station in a separate notebook cell.

Station A: price the astronomy kits

Three kits cost `19.995` credits each. The shop rule says multiply first, then round the invoice total to `0.01` with `ROUND_HALF_UP`.

```python
from decimal import Decimal, ROUND_HALF_UP

unit_price = Decimal("19.995")
quantity = 3
cent = Decimal("0.01")

raw_total = None
invoice_total = None
```

Your checks:

```python
assert raw_total == Decimal("59.985")
assert invoice_total == Decimal("59.99")
```

Station B: scale a fuel mixture

A model mixture uses fuel and stabilizer in a `5/8` ratio. A trial batch uses `14` units of stabilizer. Calculate the exact fuel amount and then a float display.

```python
from fractions import Fraction

fuel_ratio = Fraction(5, 8)
```

PYTHON (CONTINUED)

```
stabilizer_units = 14

fuel_units = None
fuel_display = None
```

Your checks:

PYTHON

```
assert fuel_units == Fraction(35, 4)
assert fuel_display == 8.75
```

Station C: locate a signal

A signal coordinate is `-3 + 4j`. Record its magnitude and conjugate.

PYTHON

```
signal = -3 + 4j

signal_distance = None
reflected_signal = None
```

Your checks:

PYTHON

```
assert signal_distance == 5.0
assert reflected_signal == -3 - 4j
```

Show the three calculations after attempting every station

PYTHON

```
from decimal import Decimal, ROUND_HALF_UP
from fractions import Fraction

unit_price = Decimal("19.995")
quantity = 3
cent = Decimal("0.01")
raw_total = unit_price * quantity
invoice_total = raw_total.quantize(cent, rounding=ROUND_HALF_UP)

fuel_ratio = Fraction(5, 8)
stabilizer_units = 14
fuel_units = fuel_ratio * stabilizer_units
fuel_display = float(fuel_units)

signal = -3 + 4j
signal_distance = abs(signal)
reflected_signal = signal.conjugate()
```

For each station, write one sentence explaining why an integer or ordinary float alone would not express the full requirement as clearly.

6. Check your representation choices

Given a new numeric requirement, ask:

1. Is the value a count, a measurement, a base-ten amount, an exact ratio, or a real/imaginary pair?
2. Did the data already pass through a float before exact construction?

3 Which rounding or denominator constraint belongs to the domain?

4 Is conversion needed only for display, or will later arithmetic use it?

5 Am I mixing representations without documenting why?

Key points

Key points

- Choose a numeric type from the rule the program must preserve.
- Construct Decimal values from decimal strings and state the rounding stage and mode.
- Fraction stores normalized integer ratios and keeps their arithmetic exact.
- Constructing Decimal or Fraction from a float carries the float’s earlier approximation forward.
- Complex values contain real and imaginary components; `abs()` gives magnitude and `.conjugate()` reverses the imaginary sign.
- Convert deliberately at a boundary rather than mixing numeric systems casually. :::

References

- Python `decimal` module
- Python `fractions.Fraction` documentation
- Python standard numeric types, including complex numbers
- Python `cmath` module for complex-number functions

Strings and Unicode Text

Online lesson and runnable notebook

Create, inspect, slice, search, clean, transform, and combine Unicode strings while understanding immutability and important text boundaries.

- **Level:** Python Foundations · Unit 2
- **Estimated time:** 3.5–5 hours
- **You will learn:** Create, inspect, slice, clean, search, transform, and combine Unicode text without assuming that every visible character has a simple representation.
- **Practice in:** Google Colab, JupyterLab, or a local editor

1. Quotation marks define a string in source code

Python strings hold text: names, labels, messages, identifiers, emoji, whitespace, and line breaks. Create ordinary strings with single or double quotes:

```python
language = "Python"
greeting = 'Hello'

print(language)
print(greeting)
print(type(language).__name__)
```

The quote characters mark the beginning and end in source code. They are not part of the displayed value.

Choose a quote style that avoids unnecessary escapes:

```python
message = "Mina's telescope is ready."
quotation = 'The display says "READY".'

print(message)
print(quotation)
```

Python treats single- and double-quoted strings the same after parsing them. Readability determines the choice.

Escape characters with a backslash

When the chosen quote appears inside the text, prefix it with `\`:

```python
message = 'Mina\'s telescope is ready.'
quotation = "The display says \"READY\"."

print(message)
print(quotation)
```

Common escape sequences include:

```python
two_lines = "north\nsouth"
tabbed = "name\tscore"
backslash = "C:\\observatory\\logs"

print(two_lines)
print(tabbed)
print(backslash)
```

`\n` is one newline character, `\t` is one tab character, and `\\` is one literal backslash.

Raw strings reduce backslash noise

A raw-string prefix makes most backslashes literal:

```python
pattern_text = r"\d{4}-\d{2}-\d{2}"
windows_folder = r"C:\observatory\logs"

print(pattern_text)
print(windows_folder)
```

Raw strings are useful for regular-expression patterns and Windows-style paths. They are not a separate runtime type:

```python
print(type(r"raw text").__name__)
```

The answer is still `str`. A raw string also cannot end with one unmatched backslash because that backslash would escape the closing quote in source code.

Triple quotes hold multiline text

```python
instructions = """Observatory checklist:
1. Open the dome.
2. Check the weather.
3. Record the start time."""

print(instructions)
```

Line breaks inside the delimiters become part of the string. Triple-quoted strings are also used as docstrings when they appear in a documented position. Unit 1 explained that placement, not the quote count alone, makes a docstring.

2. `len()` and indexes reveal a string's sequence

A string is an ordered sequence of Unicode code points. `len()` counts its current sequence elements:

```python
word = "Python"

print(len(word))
print(word[0])
print(word[1])
print(word[-1])
```

The length is `6`. Index `0` selects `P`, index `1` selects `y`, and index `-1` selects the final `n`.

Think of a nonnegative index as distance from the beginning:

```text
text:      P  y  t  h  o  n
index:     0  1  2  3  4  5
negative: -6 -5 -4 -3 -2 -1
```

An index must exist

```python
word = "Python"
print(word[10])
```

Python raises `IndexError` because the only nonnegative indexes are `0` through `5`. The error is evidence that the string is shorter than the program assumed.

An empty string has no valid character index:

```python
empty = ""
print(len(empty))
print(empty[0])
```

The first output is `0`; the second line raises `IndexError`.

Slice a range of characters

A slice uses `start:stop`, including `start` and excluding `stop`:

```python
word = "Python"

print(word[0:2])
print(word[2:6])
print(word[:3])
print(word[3:])
```

The results are `Py`, `thon`, `Pyt`, and `hon`. Omitting a bound means “from the beginning” or “through the end.”

Unlike a single index, a slice safely clips at the string boundary:

```python
word = "Python"
print(word[0:100])
```

The result is the whole word, not an error.

Add a step to a slice

```python
alphabet = "abcdefghij"

print(alphabet[::2])
print(alphabet[1::2])
print(alphabet[::-1])
```

The third slice uses a step of `-1`, producing the characters in reverse order. This is convenient for small text exercises; do not assume reversing code points always preserves the way a human writing system should display.

Checkpoint: string boundaries

String boundaries checkpoint

Trace the exact characters selected by each expression.

1. What is `"Python"[0]`?

- **A.** `"P"`
- **B.** `"y"`
- **C.** `"n"`
- **D.** `0`

2. What is `"Python"[1:4]`?

- **A.** `"Pyt"`
- **B.** `"yth"`
- **C.** `"ytho"`
- **D.** `"thon"`

3. What does index `-1` select from a non-empty string?

- **A.** The first character
- **B.** No character
- **C.** The final character
- **D.** The string length

4. What does `"cat"[:100]` produce?

- **A.** An `IndexError`
- **B.** An empty string
- **C.** One hundred characters
- **D.** `"cat"`

Write your choices before checking the answer key at the end of this book.

3. Strings are immutable

Once created, a string’s sequence of characters cannot be changed in place:

```python
word = "cat"
word[0] = "b"
```

Python raises `TypeError`: a string does not support item assignment. Build a new string instead:

```python
word = "cat"
new_word = "b" + word[1:]

print(word)
print(new_word)
```

The original remains `"cat"`; the new value is `"bat"`.

String methods follow the same pattern:

```python
name = "  mina  "
cleaned_name = name.strip()

print(repr(name))
print(repr(cleaned_name))
```

`repr()` shows quotes and otherwise invisible surrounding spaces. `.strip()` returns a new string; it does not update `name` automatically.

This common mistake discards the result:

```python
name = "  mina  "
name.strip()
print(repr(name))
```

The method ran, but no name kept its returned value. Reassign when that is the intention:

```python
name = "  mina  "
name = name.strip()
print(repr(name))
```

4. Combine and test text with small operators

`+` concatenates strings:

```python
first_name = "Ada"
last_name = "Lovelace"
full_name = first_name + " " + last_name

print(full_name)
```

`*` repeats a string an integer number of times:

```python
divider = "-" * 24
print(divider)
```

For messages that mix values, f-strings are usually clearer than repeated `+`. Lesson 6 explores their format options.

Membership asks whether text occurs inside text

```python
filename = "observations.csv"

print(".csv" in filename)
print(".json" not in filename)
```

Both results are Boolean values. Membership is case-sensitive:

```python
message = "Launch Ready"
print("ready" in message)
```

The result is `False` because lowercase `r` and uppercase `R` are different code points.

5. Clean boundaries with strip and prefix methods

External text often carries whitespace:

```python
raw_code = "\n  AX-42 \t"

print(repr(raw_code.strip()))
print(repr(raw_code.lstrip()))
print(repr(raw_code.rstrip()))
```

- `.strip()` removes whitespace from both ends;
- `.lstrip()` removes it from the left; and
- `.rstrip()` removes it from the right.

They do not remove whitespace from the middle:

```python
name = "Ada   Lovelace"
print(name.strip())
```

The internal spaces remain because they may be meaningful.

Strip removes a set of edge characters, not a word

```python
label = "xy-report-yx"
print(label.strip("xy-"))
```

`.strip("xy-")` removes any of those individual characters repeatedly from both edges. It does not look for the exact substring `"xy-"`.

Use explicit prefix/suffix methods when the contract names an exact marker:

```python
filename = "draft-observations.csv"

without_status = filename.removeprefix("draft-")
without_extension = without_status.removesuffix(".csv")

print(without_status)
```

PYTHON (CONTINUED)

```
print(without_extension)
```

If the exact prefix or suffix is absent, the method returns the original string.

Empty and whitespace-only strings are different source values

PYTHON

```
empty = ""
spaces = "   "

print(len(empty))
print(len(spaces))
print(spaces.strip() == "")
```

The raw lengths are `0` and `3`, but cleaning the spaces produces an empty string. A form rule may treat both as blank while still preserving the raw source for diagnosis.

6. Change case for display or comparison

Python provides several case transformations:

PYTHON

```
label = "mOON BASE"

print(label.lower())
print(label.upper())
print(label.capitalize())
print(label.title())
```

These methods return new strings. `.title()` can be convenient for simple display labels, but it is not a universal person-name formatter. Names and writing systems have rules that a generic casing algorithm cannot infer.

For case-insensitive matching, `.casefold()` is more aggressive and Unicode-aware than `.lower()`:

PYTHON

```
first = "Straße"
second = "STRASSE"

print(first.lower() == second.lower())
print(first.casefold() == second.casefold())
```

The casefolded comparison is `True`. Keep the original strings for display; use a normalized comparison form for matching.

Checkpoint: immutability and cleanup

String cleanup checkpoint

Choose the operation that preserves the original text and states the intended boundary.

1. After `cleaned = raw.strip()`, what happens to `raw`?

- **A.** It still refers to the original string.
- **B.** It becomes `None`.
- **C.** It is always empty.

- **D.** It changes type to a list.

2. Which expression removes the exact leading marker `"draft-"` if present?

- **A.** `name.strip("draft-")`
- **B.** `name.removeprefix("draft-")`
- **C.** `name.replace("draft-", "")`
- **D.** `name["draft-"]`

3. Why does calling `name.upper()` alone not change `name`?

- **A.** Uppercase is not Unicode.
- **B.** The method prints instead of returning.
- **C.** The method returns a new string, and the code did not bind it.
- **D.** Names cannot refer to text.

4. When is `casefold()` especially useful?

- **A.** Calculating a numeric average
- **B.** Mutating one character in place
- **C.** Encoding bytes for a file
- **D.** Preparing Unicode text for a case-insensitive comparison

Write your choices before checking the answer key at the end of this book.

7. Search, count, and replace deliberately

Use `in` for a yes/no membership question and `.find()` when the position matters:

```python
message = "weather station ready"

print("station" in message)
print(message.find("station"))
print(message.find("missing"))
```

`.find()` returns the starting index or `-1` when no match exists. Do not mistake `-1` for a valid result position in later slicing.

If absence should be an error, `.index()` has a different contract:

```python
message = "weather station ready"
print(message.index("station"))
```

Calling `.index("missing")` would raise `ValueError`. Choose between a sentinel result and an exception based on what absence means to the program.

Count non-overlapping matches

```python
report = "rain rain sun rain"
print(report.count("rain"))
print(report.count("ai"))
```

The method counts non-overlapping occurrences. It does not understand words or language boundaries; it searches exact character sequences.

Test beginnings and endings

```python
filename = "observations.csv"

print(filename.startswith("observ"))
print(filename.endswith(".csv"))
```

These methods communicate intent more directly than manually slicing and comparing a guessed number of characters.

Replace exact character sequences

```python
status = "draft: telescope draft"
updated = status.replace("draft", "ready")

print(status)
print(updated)
```

Every non-overlapping match is replaced. Limit the number when the contract says only the first occurrence:

```python
status = "draft: telescope draft"
print(status.replace("draft", "ready", 1))
```

Replacement is literal and case-sensitive; it is not a grammar-aware editing tool.

8. Split text apart and join it again

`.split()` separates text and returns a list. Lists are the next unit, so treat the container as supplied output here:

```python
raw_tags = "moon, stars, weather"
tags = raw_tags.split(",")

print(tags)
print(type(tags).__name__)
```

Spaces remain because the delimiter is only the comma. A later loop could clean each item; for now, observe the boundary.

Splitting on no explicit separator treats runs of whitespace as one boundary:

```python
sentence = "  clear   dark sky  "
words = sentence.split()
print(words)
```

The result is `['clear', 'dark', 'sky']`.

`.join()` belongs to the delimiter string:

```python
words = ["clear", "dark", "sky"]
message = " | ".join(words)
print(message)
```

The output is `clear | dark | sky`. Every item must be a string:

```python
values = ["temperature", 21.5]
print(", ".join(values))
```

Python raises `TypeError` because joining does not guess how the float should be formatted. Lesson 6 shows deliberate formatting.

9. Unicode text can have more than one code-point spelling

Python strings are Unicode text. `ord()` maps one character to its code point; `chr()` maps a code point back:

```python
symbol = "A"
code_point = ord(symbol)

print(code_point)
print(chr(code_point))
print(ord("🌙"))
```

Code points are numeric identities in the Unicode standard. They are not the same as encoded bytes; Unit 9 will teach UTF-8 and other encodings.

Visually identical text can use different sequences

The character `é` can be one code point or an `e` followed by a combining accent:

```python
composed = "é"
decomposed = "e\u0301"

print(composed)
print(decomposed)
print(composed == decomposed)
print(len(composed))
print(len(decomposed))
```

The strings usually look alike, but equality is `False` and their lengths differ. Normalize when the program’s matching rule treats canonically equivalent Unicode sequences as the same:

```python
import unicodedata

composed = "é"
decomposed = "e\u0301"

left = unicodedata.normalize("NFC", composed)
right = unicodedata.normalize("NFC", decomposed)

print(left == right)
```

NFC normalization converts both to a standard composed form where possible.

Human-visible characters are not always one string element

Some emoji and writing-system clusters contain multiple code points joined for display. `len()` counts Python string elements, not necessarily what a person perceives as one symbol. Do not design cursor movement, text truncation, or user name limits from `len()` alone without understanding the product's language and display requirements.

Normalize for matching, preserve originals for display

```python
import unicodedata

submitted = "  CAFÉ "
stored_label = "cafe\u0301"

submitted_key = unicodedata.normalize("NFC", submitted.strip()).casefold()
stored_key = unicodedata.normalize("NFC", stored_label).casefold()

print(submitted_key == stored_key)
print(submitted.strip())
```

The comparison succeeds while the original submitted capitalization remains available. A comparison key is a tool, not a replacement for the person's chosen display text.

Checkpoint: searching and Unicode

Searching and Unicode checkpoint

Choose the operation that matches the text contract.

1. Which expression asks only whether `".csv"` occurs in `filename`?

- **A.** `".csv" in filename`
- **B.** `filename.find(".csv") + 1`
- **C.** `filename[".csv"]`
- **D.** `int(filename)`

2. What does `text.find(needle)` return when `needle` is absent?

- **A.** `None`
- **B.** `-1`
- **C.** An empty list
- **D.** Always `0`

3. Why might two visually identical strings compare unequal?

- **A.** Python compares only their fonts.
- **B.** One must be an integer.
- **C.** They can contain different but canonically equivalent Unicode code-point sequences.
- **D.** Equality ignores Unicode.

4. Why does `", ".join(["score", 10])` fail?

- **A.** The delimiter is too short.
- **B.** Lists can contain only one item.
- **C.** Commas cannot appear in strings.
- **D.** `join()` requires every item to already be a string.

Write your choices before checking the answer key at the end of this book.

10. Build an observatory badge label

The badge printer receives these values:

```python
import unicodedata

raw_name = "  aDA loVELACE  "
raw_role = "guest-observer"
raw_interests = "moon,stars,weather"
comparison_name = "ada lovelace"
```

Create these names:

- `display_name` — surrounding whitespace removed and simple title casing applied;
- `display_role` — the exact `"guest-"` prefix removed, hyphens replaced by spaces, then title cased;
- `interest_text` — commas replaced by `" · "` for display;
- `name_key` — NFC-normalized, stripped, casefolded text for comparison;
- `expected_key` — the same comparison transformation for `comparison_name`;
- `same_person_text` — the string `"MATCH"` supplied below when the keys are equal; and
- `badge` — a two-line string with the name/role on the first line and interests on the second.

Because full conditional selection belongs to Unit 4, use this supplied line:

```python
same_person_text = ("NO MATCH", "MATCH")[name_key == expected_key]
```

The indexing expression selects position `1` when the Boolean comparison is `True`. Unit 3 will explain tuples; you only need the provided selection line for this lab.

Use these checks:

```python
assert raw_name == "  aDA loVELACE  "
assert display_name == "Ada Lovelace"
assert display_role == "Observer"
assert interest_text == "moon · stars · weather"
assert name_key == expected_key
assert same_person_text == "MATCH"
assert badge == (
    "Ada Lovelace — Observer\n"
    "Interests: moon · stars · weather"
)
```

Hint: build one display component at a time

Use `.strip().title()` for the simple display name. Chain `.removeprefix()`, `.replace()`, and `.title()` for the role. Normalize the stripped string before casefolding the comparison key.

Show one solution after the assertions pass

```python
import unicodedata

raw_name = "  aDA loVELACE  "
raw_role = "guest-observer"
raw_interests = "moon,stars,weather"
comparison_name = "ada lovelace"

display_name = raw_name.strip().title()
display_role = raw_role.removeprefix("guest-").replace("-", " ").title()
interest_text = raw_interests.replace(",", " · ")
name_key = unicodedata.normalize("NFC", raw_name.strip()).casefold()
expected_key = unicodedata.normalize("NFC", comparison_name).casefold()
same_person_text = ("NO MATCH", "MATCH")[name_key == expected_key]
badge = (
    f"{display_name} — {display_role}\n"
    f"Interests: {interest_text}"
)

print(badge)
print(f"Comparison: {same_person_text}")
```

Now change exactly one input:

1. Add spaces around `raw_role` and decide where `.strip()` belongs.
2. Use the decomposed string `"Ame\u0301lie"` and compare it with `"Amélie"`.
3. Remove all interests and decide what an empty display should mean. Do not invent a default silently; write the rule beside your code.

11. Check your text decisions

Explain these differences:

1. An index versus a slice beyond the end.
2. `.strip("abc")` versus `.removeprefix("abc")`.
3. Calling a string method versus storing its returned string.
4. `.lower()` versus `.casefold()` for matching.
5. Unicode text versus encoded bytes.

6 A preserved display string versus a normalized comparison key.

Key points

Key points

- Quotes delimit strings in source; escape sequences and raw strings control how backslashes are interpreted.
- Strings are ordered, support indexing and slicing, and are immutable.
- String methods return new values; keep the result when it matters.
- Choose cleanup, prefix, search, replacement, split, and join methods from the exact text boundary.
- Python strings hold Unicode text, not encoded bytes.
- Visually identical text can contain different code-point sequences; normalize when the matching contract requires it.
- Preserve a person’s original display text even when you create a normalized comparison key. :::

References

- Python tutorial: strings
- Python standard string methods
- Python lexical rules for string and bytes literals
- Python `unicodedata` module
- Unicode Standard Annex #15: normalization forms

Input, Output, and Clear Formatting

Online lesson and runnable notebook

Read console text, clean and convert it at an explicit boundary, and present values clearly with print and f-string formatting.

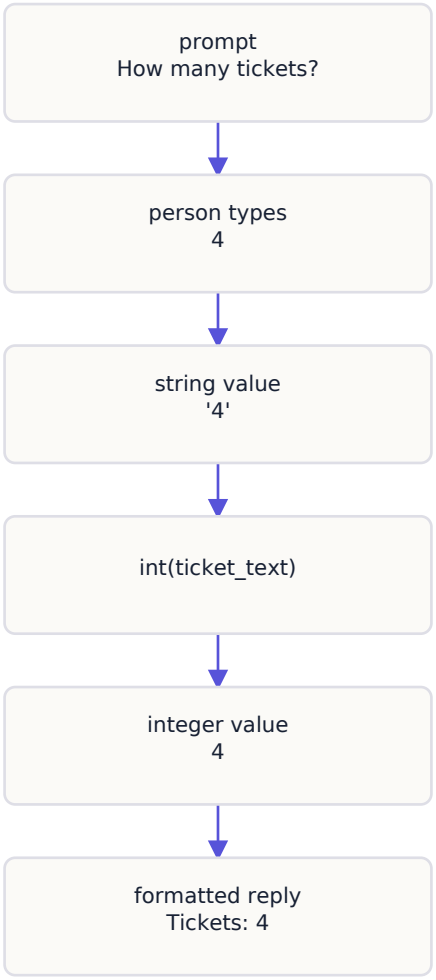
- **Level:** Python Foundations · Unit 2
- **Estimated time:** 3–4 hours
- **You will learn:** Read console input as text, convert it deliberately, and format readable human and diagnostic output.
- **Practice in:** Google Colab, JupyterLab, or a local terminal

1. A console program is a small conversation

An interactive program follows a simple path:

1. display a useful prompt;
2. receive text from the person;
3. clean or convert that text;
4. perform the calculation; and
5. display a result with enough context to understand it.

`input()` always crosses into the program as text. Conversion is a separate, visible step.



Run this cell and type your name:

```python
name = input("What is your name? ")
print(f"Hello, {name}!")
```

`input()` displays the prompt, pauses, and returns the characters typed before Enter. Assignment binds that returned string to `name`. `print()` displays the reply.

Prompts should state the expected value

Compare:

```python
value_text = input("Value: ")
```

with:

```python
temperature_text = input("Temperature in °C (for example, 21.5): ")
```

The second prompt communicates the meaning, unit, and one accepted shape. It does not guarantee valid input, but it gives the person a fair contract.

2. `print()` can display several values

`print()` accepts multiple arguments and inserts a space between them:

```python
name = "Mina"
score = 42

print("Observer", name, "scored", score)
```

Python converts each argument to display text for this output operation. The integer `score` remains an integer afterward.

Choose a separator with `sep`

```python
year = 2026
month = 8
day = 3

print(year, month, day, sep="-")
print("north", "east", "south", "west", sep=" | ")
```

`sep` is inserted **between** arguments, not after the final one.

Choose what follows with `end`

`print()` normally ends with a newline. Override it deliberately:

```python
print("Loading", end="")
print(".", end="")
print(".", end="")
print(". done")
```

The output appears on one line. Use `end` sparingly; output that never moves to a new line can be difficult to read in notebooks or terminals.

Print an empty line

```python
print("Section A")
print()
print("Section B")
```

Calling `print()` with no arguments writes only its ending newline, creating one blank line.

Display is not storage

```python
quantity = 4
print("Quantity:", quantity)
print(type(quantity).__name__)
```

The first call displays a textual representation of `quantity`. It does not replace the stored integer with text.

3. `input()` always returns a string

Run this cell and type `5`:

```python
number = input("Type a whole number: ")

print(number)
print(type(number).__name__)
print(number + number)
```

The type is `str`, and the final output is `55`. The keyboard sends characters; Python does not infer a numeric type from their appearance.

Convert at a named boundary:

```python
number_text = input("Type a whole number: ")
number = int(number_text)
doubled = number * 2

print(f"Doubled: {doubled}")
```

The suffix `_text` makes the raw representation visible. `number` holds the converted integer, and later arithmetic no longer needs to wonder whether the value is text.

Clean surrounding whitespace first

```python
city_text = input("City: ")
city = city_text.strip()

print(f"Raw: {city_text!r}")
print(f"Clean: {city!r}")
```

`!r` asks the f-string to use a diagnostic representation, so quotes and edge spaces are visible. Keep both values while investigating input problems.

Numeric constructors already accept ordinary surrounding whitespace:

```python
quantity = int("  12  ")
measurement = float("  3.5  ")

print(quantity)
print(measurement)
```

Cleaning explicitly can still make the boundary and later empty checks easier to explain.

Invalid input should fail visibly at this stage

```python
age_text = "twelve"
age = int(age_text)
```

Python raises `ValueError`. Do not replace it with a default age just to continue. Unit 8 will teach how to catch an anticipated conversion failure, display a useful message, and retry. Until then, use fixed valid values for the complete program and inspect invalid cases in separate cells.

Checkpoint: the console boundary

Console input and output checkpoint

Follow the value as it crosses the console boundary.

1. What type does `input()` return when a person types `42`?

- **A.** `str`
- **B.** `int`
- **C.** `float`
- **D.** `bool`

2. What does `print(2026, 8, 3, sep="-")` display?

- **A.** `2026 8 3`
- **B.** `2026-8-3`
- **C.** `202683`
- **D.** Three unrelated lines

3. Why use `quantity = int(quantity_text)` before `quantity * 2`?

- **A.** To erase the prompt
- **B.** To make the name uppercase
- **C.** To create an integer whose multiplication means arithmetic
- **D.** To add quotation marks

4. What does `end=""` change in a `print()` call?

- **A.** The type of every argument
- **B.** The separator between arguments
- **C.** The stored variables
- **D.** The text written after the final argument

Write your choices before checking the answer key at the end of this book.

4. F-strings place values inside readable messages

Prefix a string literal with `f` and place expressions inside braces:

```python
observer = "Mina"
objects_seen = 7

message = f"{observer} recorded {objects_seen} objects."
print(message)
```

Python evaluates each expression, converts its result for display, and constructs one new string.

Forgetting the `f` leaves braces as ordinary characters:

```python
observer = "Mina"
print("Observer: {observer}")
```

The output is literally `Observer: {observer}`. Add the prefix:

```python
print(f"Observer: {observer}")
```

Keep expressions understandable

Simple calculations can appear inside braces:

```python
completed = 8
remaining = 3

print(f"Total tasks: {completed + remaining}")
```

When an expression contains several operations or has meaning worth naming, calculate it first:

```python
completed = 8
remaining = 3
total_tasks = completed + remaining

print(f"Total tasks: {total_tasks}")
```

The second version makes `total_tasks` available for checks and later output.

Show the expression and value while debugging

The `=` debug form includes the expression text:

```python
temperature = 21.5
print(f"{temperature=}")
```

The output resembles `temperature=21.5`. This is useful while investigating, but a user-facing report should usually have a human label.

Use `!r` to reveal whitespace and escapes

```python
raw_name = "  Mina\n"

print(f"Human display: {raw_name}")
print(f"Diagnostic: {raw_name!r}")
```

The diagnostic representation shows quotes and `\n`. It is evidence for a developer, not necessarily a friendly message for the person using the program.

5. Format specifications control appearance

Place a colon after the expression to add a format specification:

```python
distance = 12.34567
print(f"Distance: {distance:.2f} km")
```

`.2f` means fixed-point notation with two digits after the decimal point. The f-string result is text; `distance` remains the original float.

Add grouping separators

```python
visitors = 1250000
budget = 9876543.21

print(f"Visitors: {visitors:,}")
print(f"Budget: {budget:,.2f}")
```

The comma improves an English-style numeric display. Localization rules vary; later applications may use a locale-aware presentation library.

Display percentages

```python
completion = 0.875
print(f"Complete: {completion:.1%}")
```

Percentage formatting multiplies by 100 for display and adds `%`, producing `87.5%`. The stored value remains `0.875`.

Align fields into columns

```python
label = "Temperature"
value = 21.5

print(f"{label:<16} | {value:>8.2f}")
print(f"{'Humidity':<16} | {0.62:>8.1%}")
```

- `<16` left-aligns text in a field 16 characters wide;
- `>8.2f` right-aligns a two-decimal number in a field eight characters wide.

Fill and center a heading:

```python
print(f"{' NIGHT REPORT ':=^32}")
```

`=` is the fill character, `^` centers, and `32` is the field width.

Display signs explicitly

```python
change = 4.2
loss = -1.7

print(f"Change: {change:+.1f}")
print(f"Change: {loss:+.1f}")
```

`+` in the specification displays a sign for both positive and negative values.

Checkpoint: formatting promises

Clear formatting checkpoint

Choose the format that makes the requested display promise.

1. Which f-string displays `price` with exactly two digits after the decimal point?

- **A.** `f"{price:.2f}"`
- **B.** `f"{price:2}"`
- **C.** `f"price.2f"`
- **D.** `f"{price}%"`

2. What does `f"{0.375:.1%}"` display?

- **A.** `0.4%`
- **B.** `37.5%`
- **C.** `375.0%`
- **D.** `0.375`

3. Which alignment marker right-aligns a value inside its field?

- **A.** `^`
- **B.** `<`
- **C.** `>`
- **D.** `=` by itself

4. After `shown = f"{value:.2f}"`, what is `shown`?

- **A.** The original float mutated in place
- **B.** An integer
- **C.** A Decimal
- **D.** A new string containing a formatted representation

Write your choices before checking the answer key at the end of this book.

6. Human output and diagnostic output serve different readers

`str()` aims for a readable representation. `repr()` aims for an unambiguous or developer-oriented representation when possible:

```python
message = "line one\nline two"

print(str(message))
print(repr(message))
```

The first output contains a real line break. The second includes quotes and the visible escape `\n`.

F-strings use the same distinction:

```python
message = "line one\nline two"

print(f"User sees:\n{message}")
print(f"Developer inspects: {message!r}")
```

Do not show raw diagnostic representations as unexplained error messages to a beginner user. Do use them while tracking invisible whitespace, newlines, and unexpected types.

Avoid formatting too early

```python
price = 3.25
price_text = f"{price:.2f}"

print(price_text)
print(type(price_text).__name__)
```

`price_text` is suitable for display but not for later numeric arithmetic. Keep `price` for calculation and create display text at the output boundary.

7. Build an interactive night-sky ticket

Create a small program that asks for:

- the visitor’s name;
- the number of tickets; and
- the price per ticket.

Use Decimal for the exact base-ten price:

```python
from decimal import Decimal

name_text = input("Visitor name: ")
quantity_text = input("Number of tickets (whole number): ")
price_text = input("Price per ticket (for example, 7.50): ")

name = None
quantity = None
unit_price = None
total = None
```

Create a three-line ticket with this shape:

```text
========== NIGHT-SKY TICKET ==========
Visitor         Mina
Tickets            3
Total          22.50 credits
```

Your exact spacing may vary, but the values must be cleaned, typed, and formatted deliberately.

Use fixed values for repeatable checks

Interactive cells are useful for a person, but assertions should not require retyping input. Put this equivalent data in a second cell:

```python
from decimal import Decimal

name_text = "  Mina  "
quantity_text = "3"
price_text = "7.50"

name = name_text.strip()
quantity = int(quantity_text)
unit_price = Decimal(price_text)
total = unit_price * quantity

assert name == "Mina"
assert quantity == 3
assert unit_price == Decimal("7.50")
assert total == Decimal("22.50")
```

Then build the output from those typed values.

Hint: give each output field a width

Try a left-aligned width for labels and a right-aligned width for values: `f"{'Visitor':<12}{name:>12}"`. Decimal supports fixed-point formatting such as `{total:.2f}`.

Show one output path after your checks pass

```python
from decimal import Decimal

name_text = "  Mina  "
quantity_text = "3"
price_text = "7.50"

name = name_text.strip()
quantity = int(quantity_text)
unit_price = Decimal(price_text)
total = unit_price * quantity

print(f"{' NIGHT-SKY TICKET ':=^38}")
print(f"{'Visitor':<14}{name:>24}")
print(f"{'Tickets':<14}{quantity:>24}")
print(f"{'Total':<14}{total:>18.2f} credits")
```

Test one invalid value in a separate cell, record its error, and restore the valid program. Do not add a `try`/`except` copied from elsewhere; exception handling will be taught with enough context in Unit 8.

8. Check your program’s conversation

Review the ticket from two perspectives:

1. Does each prompt state the meaning and expected shape?
2. Is every raw input retained as text until a visible conversion or cleanup step?
3. Are numeric calculations performed on numbers rather than formatted strings?
4. Do field widths improve scanning without removing meaningful text?

5 Can a developer reveal whitespace with `!r` without showing that diagnostic view to the visitor?

Final checkpoint

Input-to-output checkpoint

Choose the design that keeps conversion, calculation, and display separate.

1. Why keep a name such as `quantity_text` before creating `quantity`?

- **A.** It makes the external text boundary visible.
- **B.** It forces all arithmetic to concatenate.
- **C.** It prevents conversion.
- **D.** It changes the keyboard input to a float.

2. Which f-string conversion reveals edge spaces and newline escapes while debugging?

- **A.** `f"{value:%}"`
- **B.** `f"{value!r}"`
- **C.** `f"{value:,}"`
- **D.** `f"{{value}}"`

3. Which name should a later multiplication use after `total_text = f"{total:.2f}"`?

- **A.** `total_text`, because text is exact arithmetic
- **B.** Both names are automatically updated
- **C.** The numeric `total`, not its display string
- **D.** Neither; formatting deletes the value

4. Before exception handling is taught, how should this lesson investigate `int("twelve")`?

- **A.** Replace every invalid input with zero silently.
- **B.** Ignore the conversion line.
- **C.** Copy an unexplained retry loop.
- **D.** Run it separately, read the `ValueError`, record the boundary, and restore valid data.

Write your choices before checking the answer key at the end of this book.

Key points

Key points

- `input()` returns text, even when the characters look numeric.
- Clean and convert input at an explicit boundary before calculation.
- `print()` accepts multiple values and configurable `sep` and `end` text.
- F-strings produce strings and can include readable value expressions.
- Format specifications control precision, percentage, grouping, width, alignment, fill, and sign in the display.

- Formatting does not change the stored numeric value.
- Human-facing `str()` output and diagnostic `repr()` output serve different readers.
- Keep a fixed-data version of an interactive example so checks are reproducible. :::

References

- Python built-in functions: `input`, `print`, `str`, and `repr`
- Python formatted string literals
- Python format-specification mini-language
- Python tutorial: formatted string literals

Booleans, Comparisons, and Truth

Online lesson and runnable notebook

Produce and combine Boolean facts, distinguish truthiness from explicit comparison, and avoid common traps involving None, text, and numbers.

- **Level:** Python Foundations · Unit 2
- **Estimated time:** 3–4 hours
- **You will learn:** Create, name, combine, and inspect Boolean facts while choosing explicit comparisons when truthiness would hide meaning.
- **Practice in:** Google Colab, JupyterLab, or a local editor

1. A comparison produces a value

A program often needs facts before it can make decisions. Comparisons produce the two Boolean values `True` and `False`:

PYTHON
```
temperature = 18

print(temperature >= 15)
print(temperature < 10)
print(type(temperature >= 15).__name__)
```

The outputs are `True`, `False`, and `bool`. A comparison is an expression, so its result can be given a meaningful name:

PYTHON
```
temperature = 18
minimum_temperature = 15

is_warm_enough = temperature >= minimum_temperature
print(is_warm_enough)
```

`is_warm_enough` exposes the question better than repeating a numeric expression in every later output.

Six ordinary comparison operators

Operator	Meaning
==	equal values
!=	unequal values
<	less than
<=	less than or equal
>	greater than
>=	greater than or equal

Do not confuse equality with assignment:

```python
score = 80
at_target = score == 80

print(score)
print(at_target)
```

The single `=` binds a name. The double `==` compares two values and produces a Boolean result.

Boundary symbols change the accepted set

```python
age = 18

print(age > 18)
print(age >= 18)
```

At exactly 18, the first comparison is `False` and the second is `True`. Translate “older than” and “at least” carefully; one character changes the rule.

Chain a range comparison

Python can express a closed range directly:

```python
score = 84
minimum = 60
maximum = 100

in_range = minimum <= score <= maximum
print(in_range)
```

This means `score` is at least `minimum` **and** at most `maximum`. It is usually clearer than writing both comparisons separately.

Test the boundaries:

```python
print(60 <= 60 <= 100)
print(60 <= 100 <= 100)
print(60 <= 101 <= 100)
```

The inclusive endpoints are accepted; `101` is not.

2. Compare compatible meanings, not merely compatible syntax

Python can compare integers and floats numerically:

```python
print(4 == 4.0)
print(3 < 3.5)
```

Both are `True`. The types differ, but they participate in a common numeric comparison.

Boolean values have a historical numeric relationship:

```python
print(True == 1)
print(False == 0)
print(True + True)
```

The results are `True`, `True`, and `2`. Recognize this behavior, but do not use Boolean arithmetic when the program means a count. Give counts integer names and facts Boolean names.

String ordering is lexicographic by Unicode code points

```python
print("apple" < "banana")
print("Zoo" < "apple")
print("10" < "2")
```

Python compares the first differing character. Uppercase and lowercase code points have their own order, and digit strings are compared as text. `"10" < "2"` is `True` because the first character `"1"` comes before `"2"`.

Convert numeric text before a numeric comparison:

```python
left_text = "10"
right_text = "2"

print(int(left_text) < int(right_text))
```

The result is `False`, matching the numeric question.

Unrelated ordering raises an error

```python
print("10" < 2)
```

Python raises `TypeError` rather than guessing whether to compare text or numbers. Equality has a different contract and normally returns `False` for such unrelated values:

```python
print("10" == 10)
```

That does not mean the values are “nearly equal.” It means they are not equal under Python’s value rules.

Float comparisons need the precision contract

```python
import math

calculated = 0.1 + 0.2

print(calculated == 0.3)
print(math.isclose(calculated, 0.3))
```

Lesson 3 established the distinction: exact equality and closeness answer different questions.

Checkpoint: comparison facts

Comparison facts checkpoint

Translate the exact boundary and value meanings.

1. Which expression means `age` is at least 18?

- **A.** `age >= 18`
- **B.** `age > 18`
- **C.** `age == 17`
- **D.** `age = 18`

2. Which expression accepts every `score` from 0 through 100, including both endpoints?

- **A.** `score < 0 > 100`
- **B.** `0 <= score <= 100`
- **C.** `score == 0 == 100`
- **D.** `0 < 100 < score`

3. Why is `"10" < "2"` true?

- **A.** Python adds the digits.
- **B.** Ten is numerically smaller.
- **C.** Strings compare character order, and `1` comes before `2`.
- **D.** Every non-empty string is less than another.

4. Which comparison states that two measurements may differ within a tolerance?

- **A.** `left is right`
- **B.** `str(left) == str(right)`
- **C.** `left = right`
- **D.** `math.isclose(left, right, ...)`

Write your choices before checking the answer key at the end of this book.

3. Truthiness asks whether a value counts as false in context

`bool(value)` reveals a value's truthiness:

```python
print(bool(0))
print(bool(7))
print(bool(-2))
print(bool(0.0))
```

Numeric zero is falsy. Other numbers are truthy, including negative numbers.

Strings follow emptiness:

```python
print(bool(""))
print(bool("hello"))
print(bool(" "))
print(bool("False"))
```

Only the empty string is falsy. A space is a character, and the word `"False"` is non-empty text; both are truthy.

`None` is falsy:

```python
print(bool(None))
```

Collections use emptiness too. These are previews of Unit 3:

```python
print(bool([]))
print(bool([0]))
print(bool({}))
print(bool({"ready": False}))
```

An empty list or dictionary is falsy. A collection containing a falsy item is still non-empty and therefore truthy.

Truthiness can collapse different meanings

```python
unknown_score = None
zero_score = 0
blank_score = ""

print(bool(unknown_score))
print(bool(zero_score))
print(bool(blank_score))
```

All three are falsy, but they can mean “not measured,” “measured as zero,” and “submitted as blank.” Use an explicit comparison when the distinction matters:

```python
score = 0

has_score = score is not None
score_is_positive = score > 0

print(has_score)
print(score_is_positive)
```

The program has a score, but it is not positive.

Parse human Boolean text with an explicit rule

This is wrong for yes/no input:

```python
answer_text = "no"
answer = bool(answer_text)
print(answer)
```

It prints `True`. Instead, clean and compare the accepted spelling:

```python
answer_text = "  YES "
answer = answer_text.strip().casefold() == "yes"
print(answer)
```

This simple rule recognizes only `yes` as true; every other spelling produces `False`. A production form may need to reject unknown answers rather than treating them as no. Exception and validation units will design that boundary explicitly.

Checkpoint: truthiness versus meaning

Truthiness checkpoint

Distinguish emptiness from a domain-specific fact.

1. What is `bool(0)`?

- **A.** `False`
- **B.** `True`
- **C.** `None`
- **D.** A `TypeError`

2. What is `bool(" " )` for a string containing one space?

- **A.** `False` because spaces are invisible
- **B.** `True` because the string is non-empty
- **C.** `None`
- **D.** A `ValueError`

3. Why is `bool("False")` true?

- **A.** Python interprets the English word backwards.
- **B.** `False` is an integer.
- **C.** The string contains characters and is therefore non-empty.
- **D.** Every string is false.

4. Which check distinguishes a real score of zero from a missing score?

- **A.** `bool(score)`
- **B.** `not score`
- **C.** `score == False`
- **D.** `score is not None`

Write your choices before checking the answer key at the end of this book.

4. Combine facts with `not`, `and`, and `or`

`not` reverses truthiness:

```python
is_cloudy = False
is_clear = not is_cloudy

print(is_clear)
```

`and` requires both sides to be truthy:

```python
has_ticket = True
has_id = True

can_enter = has_ticket and has_id
print(can_enter)
```

`or` requires at least one side to be truthy:

```python
has_member_pass = False
has_day_ticket = True

has_admission = has_member_pass or has_day_ticket
print(has_admission)
```

Name intermediate facts

```python
age = 17
has_permission = True
has_ticket = True

is_adult = age >= 18
age_requirement_met = is_adult or has_permission
can_enter = age_requirement_met and has_ticket

print(is_adult)
print(age_requirement_met)
print(can_enter)
```

The names show the rule in parts. A reader can inspect which fact changed instead of decoding one long expression.

Precedence is `not`, then `and`, then `or`

```python
result = True or False and False
print(result)
```

`and` runs first, so the expression is `True or False`, which is `True`. Parentheses can state a different rule:

```python
result = (True or False) and False
print(result)
```

Now the result is `False`. Prefer named facts or parentheses whenever a reader could reasonably group the rule another way.

Short-circuiting can skip the right operand

For `and`, a falsy left side already determines the result:

```python
quantity = 0
safe_to_divide = quantity != 0 and 100 / quantity > 2

print(safe_to_divide)
```

Python does not evaluate `100 / quantity`, so no division-by-zero error occurs.

For `or`, a truthy left side already determines the result:

```python
has_ticket = True
allowed = has_ticket or undefined_backup_pass

print(allowed)
```

The undefined name is not evaluated because `has_ticket` is already truthy. Short-circuiting is useful, but Unit 4 will teach how to design larger decision rules without hiding unsafe operations.

5. `and` and `or` return operands, not always Booleans

Try these expressions:

```python
print("Ada" and "ready")
print("" and "ready")
print("Ada" or "anonymous")
print("" or "anonymous")
```

The results are `"ready"`, `""`, `"Ada"`, and `"anonymous"`.

- `a and b` returns the first falsy operand, or the final operand if all are truthy.
- `a or b` returns the first truthy operand, or the final operand if all are falsy.

This explains the earlier optional-text pattern:

```python
coupon_text = "   "
coupon_code = coupon_text.strip() or None

print(coupon_code)
print(coupon_code is None)
```

The cleaned empty string is falsy, so `or` returns `None`.

Use this behavior for a deliberate fallback, not as a substitute for every validation rule. A legitimate value such as zero would also trigger an `or` fallback:

```python
recorded_score = 0
display_score = recorded_score or "not recorded"
print(display_score)
```

The output is misleading because zero is a real score. An explicit `is None` decision is required; Unit 4 will implement it with a conditional.

Checkpoint: Boolean operators

Boolean operators checkpoint

Trace evaluation order and the operand each expression returns.

1. What is `not False`?

- **A.** `True`
- **B.** `False`
- **C.** `None`
- **D.** `0` only

2. When are two Boolean facts `left and right` true?

- **A.** When either one is true
- **B.** Only when both are true
- **C.** Only when both are false
- **D.** Whenever `left` is false

3. What does `"" or "anonymous"` return?

- **A.** `False`
- **B.** The empty string
- **C.** The string `"anonymous"`
- **D.** A `TypeError`

4. Why does `False and (1 / 0)` not raise `ZeroDivisionError`?

- **A.** Division by zero equals false.
- **B.** Parentheses disable division.
- **C.** `and` converts zero to one.
- **D.** A falsy left operand determines the result, so the right operand is skipped.

Write your choices before checking the answer key at the end of this book.

6. Use `is` only for the identity question you mean

For missing values, use:

```python
review_date = None

print(review_date is None)
print(review_date is not None)
```

Do not use `is` as a substitute for equality with ordinary numbers or strings:

```python
left = "moon"
right = "".join(["m", "o", "o", "n"])

print(left == right)
print(left is right)
```

The values compare equal, but identity is an implementation-level object question and may be false. Unit 6 will teach references and identity. Until then:

- use `==` and `!=` for ordinary value equality;
- use `is None` and `is not None` for the missing-value singleton.

7. Build an observatory readiness panel

Use these fixed values:

```python
temperature_c = 12.5
wind_kph = 18
cloud_cover = 0.20
dome_status = "OPEN"
operator_name = "  Mina  "
emergency_stop = False
```

The observatory can begin a viewing session when all of these facts are true:

- temperature is from `-10` through `30` °C;
- wind is no more than `25` km/h;
- cloud cover is below `0.35`;
- cleaned dome status equals `"open"` without case sensitivity;
- the cleaned operator name is non-empty; and
- the emergency stop is not active.

Create these names:

```python
temperature_ok = None
wind_ok = None
sky_clear_enough = None
dome_open = None
operator_present = None
emergency_clear = None
ready = None
```

Then run:

```python
assert temperature_ok is True
assert wind_ok is True
assert sky_clear_enough is True
assert dome_open is True
assert operator_present is True
assert emergency_clear is True
assert ready is True
```

Print every named fact before the combined result. If `ready` becomes false after a modification, the individual facts should reveal why.

Hint: translate each bullet before combining them

Use a chained comparison for temperature, `<=` for wind, `<` for cloud cover, `.strip().casefold()` for text, `bool(operator_name.strip())` for non-empty cleaned text, and `not` for the emergency stop.

Show one solution after checking every named fact

PYTHON

```
temperature_c = 12.5
wind_kph = 18
cloud_cover = 0.20
dome_status = "OPEN"
operator_name = "  Mina  "
emergency_stop = False

temperature_ok = -10 <= temperature_c <= 30
wind_ok = wind_kph <= 25
sky_clear_enough = cloud_cover < 0.35
dome_open = dome_status.strip().casefold() == "open"
operator_present = bool(operator_name.strip())
emergency_clear = not emergency_stop
ready = (
    temperature_ok
    and wind_ok
    and sky_clear_enough
    and dome_open
    and operator_present
    and emergency_clear
)

print(f"Temperature acceptable: {temperature_ok}")
print(f"Wind acceptable:        {wind_ok}")
print(f"Sky clear enough:       {sky_clear_enough}")
print(f"Dome open:              {dome_open}")
print(f"Operator present:       {operator_present}")
print(f"Emergency clear:        {emergency_clear}")
print(f"Ready to observe:       {ready}")
```

Change one boundary at a time:

1. Set `wind_kph` to exactly `25`, then `26`.
2. Set `operator_name` to three spaces.
3. Set `dome_status` to `" closed "`.
4. Set `emergency_stop` to `True`.

Predict the individual fact and final readiness before each clean rerun.

8. Check the meaning of every fact

Before moving on, explain:

1. Why `bool("False")` does not parse a human answer.
2. When a chained comparison includes both boundaries.
3. Why `score is not None` differs from `bool(score)` for zero.
4. Which operand short-circuiting can skip.

5 Why `a or fallback` can mishandle a legitimate zero.

6 Why equality, not identity, compares ordinary string values.

Key points

Key points

- Comparisons produce Boolean values that can be named and inspected.
- Boundary operators such as `<` and `<=` express different accepted sets.
- String ordering compares characters, not numeric meaning.
- Truthiness collapses several empty or zero-like values; explicit comparisons preserve distinctions.
- `bool()` does not parse human yes/no text.
- `not`, `and`, and `or` combine facts and short-circuit evaluation.
- `and` and `or` return operands, so fallbacks must respect legitimate falsy values.
- Use `==` for ordinary value equality and `is None` for missing-value identity. :::

References

- Python standard types: truth-value testing
- Python reference: comparisons
- Python reference: Boolean operations
- PEP 8 guidance for Boolean and `None` comparisons

Dates, Times, and Durations

Online lesson and runnable notebook

Represent calendar dates, clock times, timestamps, and durations with datetime types while parsing, formatting, and retaining timezone meaning.

- **Level:** Python Foundations · Unit 2
- **Estimated time:** 3–4.5 hours
- **You will learn:** Choose among `date`, `time`, `datetime`, and `timedelta`; parse and format timestamps; calculate calendar boundaries; and recognize timezone-aware values.
- **Practice in:** Google Colab, JupyterLab, or a local editor

1. Calendar text is not yet a calendar value

These strings look like dates:

```python
first_text = "2026-08-03"
second_text = "2026-08-14"

print(first_text)
print(second_text)
print(type(first_text).__name__)
```

They are still `str` values. ISO-formatted dates happen to sort sensibly as text, but text cannot reliably validate leap days, add durations, or expose calendar parts.

Python’s `datetime` standard-library module supplies four core value types:

Type	Represents	Example question
`date`	calendar year, month, and day	Which day is the visit?
`time`	wall-clock time without a date	What time does the gate open?
`datetime`	date and time together	When did the event occur?
`timedelta`	elapsed duration	How long until review?

Construct them directly:

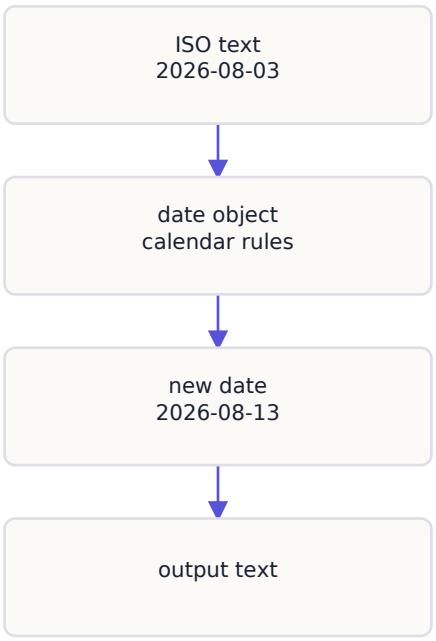
```python
from datetime import date, datetime, time, timedelta

visit_day = date(2026, 8, 3)
gate_opens = time(19, 30)
check_in = datetime(2026, 8, 3, 19, 15)
session_length = timedelta(hours=2, minutes=30)

print(visit_day)
print(gate_opens)
print(check_in)
print(session_length)
```

The constructor arguments are integers, not human-formatted strings.

Parsing crosses from external text into a value with calendar behavior. Formatting crosses back to text for a person or external system.



ISO text / 2026-08-03 → date object / calendar rules: fromisoformat

date object / calendar rules → new date / 2026-08-13: + timedelta

new date / 2026-08-13 → output text: isoformat / strftime

Inspect calendar components

PYTHON
```
from datetime import date, datetime

visit_day = date(2026, 8, 3)
check_in = datetime(2026, 8, 3, 19, 15, 30)

print(visit_day.year)
print(visit_day.month)
print(visit_day.day)
print(check_in.hour)
print(check_in.minute)
print(check_in.second)
```

These attributes are integers. They let a program reason about parts without slicing guessed character positions.

2. ISO text provides a dependable simple boundary

ISO 8601 places larger units before smaller ones:

TEXT
```
date:      YYYY-MM-DD
datetime:  YYYY-MM-DDTHH:MM:SS+offset
```

Parse with `fromisoformat()`:

```python
from datetime import date, datetime, time

visit_day = date.fromisoformat("2026-08-03")
gate_opens = time.fromisoformat("19:30:00")
check_in = datetime.fromisoformat("2026-08-03T19:15:30")

print(visit_day)
print(gate_opens)
print(check_in)
```

The result types now support date/time operations.

Return to stable ISO text

```python
from datetime import date, datetime

visit_day = date(2026, 8, 3)
check_in = datetime(2026, 8, 3, 19, 15, 30)

print(visit_day.isoformat())
print(check_in.isoformat())
```

`isoformat()` produces strings. Keep the objects for calculation; format at an output boundary.

Invalid calendar values fail during construction

Run each case separately:

```python
from datetime import date

impossible = date(2026, 2, 30)
```

```python
from datetime import date

impossible = date.fromisoformat("2026-13-01")
```

Both raise `ValueError`. The type enforces month/day relationships instead of storing an impossible date. Exception handling comes later; preserve and read the message now.

Leap years are part of the calendar rule

```python
from datetime import date

leap_day = date.fromisoformat("2024-02-29")
print(leap_day)
```

Change the year to `2023` and predict the failure. Do not implement leap-year logic by manually checking divisibility; the date constructor already owns that calendar rule.

Checkpoint: choosing and parsing types

Date and time types checkpoint

Choose the value type or boundary operation that matches the requirement.

1. Which type represents a calendar day without a clock time?

- **A.** `date`
- **B.** `time`
- **C.** `timedelta`
- **D.** `str` only

2. Which type represents an elapsed span of 90 minutes?

- **A.** `date`
- **B.** `timedelta`
- **C.** `time`
- **D.** `bool`

3. Which expression parses ISO date text into a calendar value?

- **A.** `str("2026-08-03")`
- **B.** `time("2026-08-03")`
- **C.** `date.fromisoformat("2026-08-03")`
- **D.** `int("2026-08-03")`

4. What happens when `date(2026, 2, 30)` runs?

- **A.** It silently changes to March 2.
- **B.** It creates text.
- **C.** It returns `None`.
- **D.** It raises `ValueError` because the calendar date is impossible.

Write your choices before checking the answer key at the end of this book.

3. `timedelta` represents elapsed duration

Add a duration to a date:

```python
from datetime import date, timedelta

start = date(2026, 8, 3)
review_delay = timedelta(days=10)
review_day = start + review_delay

print(review_day)
```

The result is August 13. Python handles the month or year boundary when necessary:

```python
from datetime import date, timedelta

start = date(2026, 12, 28)
review_day = start + timedelta(days=10)

print(review_day)
```

The result is in 2027 without any manual month calculation.

Subtract calendar values to obtain a duration

```python
from datetime import date

arrival = date(2026, 8, 14)
departure = date(2026, 8, 3)
stay = arrival - departure

print(stay)
print(stay.days)
print(type(stay).__name__)
```

The difference is a `timedelta` whose `.days` attribute is `11`.

Durations can include smaller units

```python
from datetime import timedelta

duration = timedelta(days=1, hours=2, minutes=30, seconds=15)

print(duration)
print(duration.total_seconds())
```

`.total_seconds()` includes every component. The `.seconds` attribute does **not** mean the same thing:

```python
from datetime import timedelta

duration = timedelta(days=1, seconds=30)

print(duration.days)
print(duration.seconds)
print(duration.total_seconds())
```

The results are `1`, `30`, and `86430.0`. `.seconds` is the leftover seconds after whole days, not the entire duration converted to seconds.

Negative durations keep normalized parts

```python
from datetime import timedelta

duration = timedelta(seconds=-1)
print(duration)
print(duration.days)
print(duration.seconds)
print(duration.total_seconds())
```

Python may display `-1 day, 23:59:59`. The normalized `.days` and `.seconds` components look surprising, while `.total_seconds()` clearly reports `-1.0`.

Thirty days is not one calendar month

PYTHON
```
from datetime import date, timedelta

start = date(2026, 1, 31)
later = start + timedelta(days=30)

print(later)
```

This calculates exactly 30 elapsed days. It does **not** implement “the same day next month,” which is ambiguous when the next month lacks day 31. If a requirement says “one calendar month,” obtain a precise business rule or use an appropriate calendar-aware tool later. Do not silently substitute 30 days.

Compare dates directly

PYTHON
```
from datetime import date

today_for_example = date(2026, 8, 3)
deadline = date(2026, 8, 14)

print(today_for_example < deadline)
print(today_for_example == deadline)
```

Fixed dates make this example reproducible. `date.today()` is useful in real code, but an assertion tied to the learner’s current day will eventually fail.

4. Custom text formats need an explicit format contract

Not every external system sends ISO text. Parse a known custom format with `datetime.strptime()`:

PYTHON
```
from datetime import datetime

text = "03/08/2026 19:30"
moment = datetime.strptime(text, "%d/%m/%Y %H:%M")

print(moment)
```

The format codes describe the input:

Code	Meaning in this example
`%d`	zero-padded day
`%m`	zero-padded month
`%Y`	four-digit year
`%H`	24-hour clock hour
`%M`	minute

The exact punctuation and spaces are part of the contract.

Ambiguous human dates require a policy

The text `03/08/2026` could mean 3 August or March 8. Python cannot infer the sender's locale. The format string above explicitly chooses day/month/year. Prefer ISO text between systems whenever you control both ends.

Format a calendar value for a person

```python
from datetime import datetime

moment = datetime(2026, 8, 3, 19, 30)

print(moment.strftime("%Y-%m-%d %H:%M"))
print(moment.strftime("%A, %d %B %Y at %H:%M"))
```

`strftime()` returns a string. Names such as Monday and August can depend on the process locale, so do not assert English month names in a multilingual environment unless locale is part of the setup.

A mismatched format fails visibly

```python
from datetime import datetime

moment = datetime.strptime("2026-08-03", "%d/%m/%Y")
```

Python raises `ValueError` because the characters do not match the declared format. Fix the source or format contract; do not keep adding formats without knowing which input the system promises.

Checkpoint: durations and formats

Duration and parsing checkpoint

Choose the operation that preserves elapsed and calendar meaning.

1. What type results from subtracting one `date` from another?

- **A.** `timedelta`
- **B.** `time`
- **C.** `str`
- **D.** `bool`

2. Which expression includes both whole days and leftover seconds in one duration total?

- **A.** `duration.seconds`
- **B.** `duration.total_seconds()`
- **C.** `duration.day`
- **D.** `str(duration).seconds`

3. What does the format string passed to `strptime()` describe?

- **A.** Only the desired output color

- **B.** The computer's current date
- **C.** The expected fields, order, punctuation, and spacing of the input text
- **D.** A duration to add

4. Why is `timedelta(days=30)` not automatically "one calendar month"?

- **A.** A timedelta cannot contain days.
- **B.** Every month has 30 days.
- **C.** Dates cannot cross months.
- **D.** Calendar months have different lengths and end-of-month rules need a policy.

Write your choices before checking the answer key at the end of this book.

5. A timestamp needs a timezone policy

This datetime has no offset or timezone information:

```python
from datetime import datetime

naive = datetime.fromisoformat("2026-08-03T19:30:00")

print(naive)
print(naive.tzinfo)
```

Python calls it **naive**. It may represent a local wall-clock time, but the value alone cannot identify one instant worldwide.

Parse an explicit UTC offset:

```python
from datetime import datetime

aware = datetime.fromisoformat("2026-08-03T19:30:00+00:00")

print(aware)
print(aware.tzinfo)
print(aware.utcoffset())
```

This value is **aware**. The `+00:00` offset identifies UTC at that moment.

Construct an aware UTC datetime directly

```python
from datetime import datetime, timezone

moment = datetime(2026, 8, 3, 19, 30, tzinfo=timezone.utc)

print(moment.isoformat())
```

The ISO output includes `+00:00`.

Convert an instant to a named region

The standard-library `zoneinfo` module uses IANA timezone rules:

```python
from datetime import datetime
from zoneinfo import ZoneInfo

utc_moment = datetime.fromisoformat("2026-08-03T19:30:00+00:00")
new_york = utc_moment.astimezone(ZoneInfo("America/New_York"))

print(utc_moment.isoformat())
print(new_york.isoformat())
```

The wall-clock hour changes, but both values describe the same instant. A named zone carries historical and daylight-saving rules that a fixed offset alone does not.

Some minimal systems may not include timezone data. If `ZoneInfo` reports missing data locally, use the Colab environment for this example and treat timezone data as a declared runtime dependency in a real project.

Do not compare naive and aware datetimes

```python
from datetime import datetime

naive = datetime.fromisoformat("2026-08-03T19:30:00")
aware = datetime.fromisoformat("2026-08-03T19:30:00+00:00")

print(naive < aware)
```

Python raises `TypeError` for ordering because the naive value has no location on the global timeline. Do not fix this by attaching an arbitrary timezone. Find the source’s intended timezone policy.

Current time is useful but not deterministic

```python
from datetime import datetime, timezone

current_utc = datetime.now(timezone.utc)
print(current_utc.isoformat())
```

This is appropriate when the task truly needs the current instant. Lessons and assertions should use fixed values so they remain reproducible tomorrow and in another region.

Checkpoint: timezone meaning

Timezone meaning checkpoint

Choose the value or operation that preserves the instant’s meaning.

1. Which ISO text contains explicit timezone information?

- **A.** `2026-08-03T19:30:00+00:00`
- **B.** `2026-08-03T19:30:00`
- **C.** `2026-08-03`
- **D.** `19:30:00`

2. Which method converts one aware instant for display in another timezone?

- **A.** `.strip()`

- **B.** `.astimezone(...)`
- **C.** `.total_seconds()`
- **D.** `.removeprefix()`

3. What information is missing from a naive datetime?

- **A.** Its year
- **B.** Its minute
- **C.** The offset or timezone policy needed to place it on a global timeline
- **D.** Its Python type

4. Why should a lesson assertion prefer a fixed datetime over `datetime.now()`?

- **A.** Fixed dates cannot be parsed.
- **B.** Current time is always naive.
- **C.** A fixed value uses less source code.
- **D.** The expected result stays reproducible across runs and locations.

Write your choices before checking the answer key at the end of this book.

6. Build an aware observatory booking

An observatory receives this fixed booking data:

PYTHON
```
from datetime import datetime, timedelta
from zoneinfo import ZoneInfo

visitor = "Mina"
starts_at_text = "2026-08-03T23:30:00+00:00"
session_minutes_text = "150"
reminder_days_text = "2"
display_zone_name = "America/New_York"

starts_at = None
session_length = None
ends_at = None
reminder_at = None
local_start = None
summary = None
```

Create an aware timestamp from the ISO text, convert the duration strings to integers, calculate the end and reminder instants, and convert the start to the named display zone.

Build this stable summary from the UTC-aware values:

TEXT
```
Mina: 2026-08-03 23:30 UTC → 2026-08-04 02:00 UTC
Reminder: 2026-08-01 23:30 UTC
```

Use these assertions:

PYTHON
```
from datetime import datetime, timedelta, timezone
```

PYTHON (CONTINUED)

```
assert starts_at == datetime(2026, 8, 3, 23, 30, tzinfo=timezone.utc)
assert session_length == timedelta(minutes=150)
assert ends_at == datetime(2026, 8, 4, 2, 0, tzinfo=timezone.utc)
assert reminder_at == datetime(2026, 8, 1, 23, 30, tzinfo=timezone.utc)
assert local_start.tzinfo is not None
assert local_start.hour == 19
assert summary == (
    "Mina: 2026-08-03 23:30 UTC → 2026-08-04 02:00 UTC\n"
    "Reminder: 2026-08-01 23:30 UTC"
)
```

Hint: keep arithmetic in aware datetime values

Parse with `datetime.fromisoformat()`. Create timedeltas from the converted minute and day counts. Use `.astimezone(ZoneInfo(display_zone_name))` only for the local display value. Format the stable summary with `strftime("%Y-%m-%d %H:%M UTC")` because its source values are explicitly UTC.

Show one solution after the booking checks pass

PYTHON

```
from datetime import datetime, timedelta
from zoneinfo import ZoneInfo

visitor = "Mina"
starts_at_text = "2026-08-03T23:30:00+00:00"
session_minutes_text = "150"
reminder_days_text = "2"
display_zone_name = "America/New_York"

starts_at = datetime.fromisoformat(starts_at_text)
session_length = timedelta(minutes=int(session_minutes_text))
ends_at = starts_at + session_length
reminder_at = starts_at - timedelta(days=int(reminder_days_text))
local_start = starts_at.astimezone(ZoneInfo(display_zone_name))
summary = (
    f"{visitor}: {starts_at:%Y-%m-%d %H:%M UTC} → "
    f"{ends_at:%Y-%m-%d %H:%M UTC}\n"
    f"Reminder: {reminder_at:%Y-%m-%d %H:%M UTC}"
)

print(summary)
print(f"Local start: {local_start.isoformat()}")
```

Change the start to `2026-12-31T23:30:00+00:00` and predict the end date. Then change `display_zone_name` to `"Asia/Tokyo"` and explain why the local calendar date may differ even though the instant has not changed.

7. Check every time boundary

Before leaving the unit, explain:

1. Why ISO date text must be parsed before calendar arithmetic.
2. The difference between `.seconds` and `.total_seconds()` on a timedelta.
3. Why 30 days is not a universal calendar-month rule.
4. What a custom `strptime()` format promises.
5. What makes a datetime aware.
6. Why `astimezone()` changes the display fields but not the instant.

Key points

Key points

- `date`, `time`, `datetime`, and `timedelta` represent different temporal ideas.
- Parse supported ISO text with `fromisoformat()` and return stable ISO text with `isoformat()`.
- Calendar constructors reject impossible dates.
- Date subtraction produces a duration; adding a duration crosses calendar boundaries safely.
- `.total_seconds()` covers the complete duration, unlike the normalized `.seconds` component.
- `strptime()` and `strftime()` use an explicit text-format contract.
- An aware datetime has an offset or timezone policy; a naive datetime cannot identify one global instant by itself.
- Use fixed timestamps for reproducible examples and assertions. :::

References

- Python `datetime` module
- Python `zoneinfo` module
- Python `strftime()` and `strptime()` format codes
- Python documentation: aware and naive datetime objects

Unit Challenge: Build a Date-Aware Purchase Summary

Online lesson and runnable notebook

Repair a time-travel supply record by converting raw text, choosing numeric representations, checking facts, calculating an aware review date, and formatting a clearance message.

- **Level:** Python Foundations · Unit 2 challenge
- **Estimated time:** 60–120 minutes
- **Outcome assessed:** Choose, convert, compare, calculate, and format text, numeric, Boolean, missing, and timezone-aware values in one reproducible artifact.
- **Evidence:** Progressive checks, a decoded clearance message, one boundary rerun, and one debugging record

1. Challenge outcome

The Chrononaut Supply Kiosk has recovered a purchase record from a damaged time capsule. Every field survived as text, several values carry extra whitespace, and the cargo scanner stored the familiar floating-point result of `0.1 + 0.2`.

Repair the record and unlock its clearance message. Your program must preserve the raw evidence, create values with deliberate types, calculate exact credit totals, confirm the measured cargo mass with a tolerance, retain the timestamp’s UTC offset, derive a small launch code, and produce one stable summary.

The challenge is a puzzle, but its rules are all on this page. You do not need collections, loops, functions, conditional statements, or exception handlers.

Mission rule

Run one assertion group at a time. When a group fails, read the first failure and repair the value that group describes. Do not edit expected values merely to open the next stage. :::

2. Understand the acceptance example

The repaired record must produce exactly this clearance message:

```text
=== CHRONONAUT SUPPLY CLEARANCE ===
Traveler: Ada Byron [AS0803]
Cargo: 4 × Quantum Compass
Credits: 13.00 + 0.98 = 13.98
Mass check: PASS | Priority: True
Review: 2026-09-02 14:30 UTC
```

The launch code is not random:

- `A` is the first character of the cleaned traveler name;
- `S` is the final character of the cleaned item name; and

- `0803` is the purchase month and day.

Before writing code, identify which displayed values should remain strings, which should become `int`, `float`, `Decimal`, `bool`, `None`, or `datetime`, and which values exist only for formatted output.

Constraints

1. Preserve every supplied name ending in `_text` unchanged.
2. Keep the supplied public names so the progressive assertions can inspect them.
3. Construct `Decimal` values directly from text, not through `float`.
4. Use `ROUND_HALF_UP` and quantize the service fee to `Decimal("0.01")`.
5. Parse priority by cleaning and comparing text; do not use `bool(priority_text)`.
6. Represent the blank coupon code with `None` and check it with `is None`.
7. Use `math.isclose()` with the supplied absolute tolerance for the mass check.
8. Parse the aware ISO timestamp rather than slicing its date characters.
9. Derive the launch code from the cleaned values and parsed timestamp.
10. Do not hard-code calculated totals, the review timestamp, code, status, or complete summary.

3. Start from the contract

Copy this starter into a fresh cell. The names set to `None` are the values you must calculate.

PYTHON

```
import math
import unicodedata
from datetime import datetime, timedelta
from decimal import Decimal, ROUND_HALF_UP

# Recovered external fields: preserve these exact source values.
traveler_text = "  aDA BYRON  "
item_text = "quantum compass"
quantity_text = "4"
unit_price_text = "3.25"
service_rate_text = "0.075"
mass_text = "0.30000000000000004"
priority_text = " YES "
coupon_text = "   "
purchased_at_text = "2026-08-03T14:30:00+00:00"
review_days_text = "30"

# Supplied scanner policy.
expected_mass = 0.3
mass_tolerance = 0.000001
cent = Decimal("0.01")

# Stage A: cleaned and converted values.
traveler = None
traveler_key = None
item = None
quantity = None
unit_price = None
service_rate = None
measured_mass = None
is_priority = None
coupon_code = None
purchased_at = None
```

PYTHON (CONTINUED)

```
review_days = None

# Stage B: calculated mission facts.
subtotal = None
service_fee = None
total = None
mass_matches = None
mass_status = None
review_at = None
launch_code = None
record_ready = None

# Stage C: final artifact.
summary = None
```

4. Build in small stages

Stage A: restore typed values

1. Strip and title-case the traveler and item display values.
2. Build `traveler_key` by stripping, NFC-normalizing, and casefolding the raw traveler text. This key is evidence that you can prepare text for matching without replacing the display name.
3. Convert the quantity and review-day count to integers.
4. Construct the price and service rate as Decimals from their source strings.
5. Convert the measured mass to float.
6. Parse priority with an explicit cleaned-text comparison.
7. Clean the coupon text and use `or None` to represent its absence.
8. Parse the ISO timestamp with `datetime.fromisoformat()`.

Run assertion group A before continuing.

Stage B: recover the purchase and time facts

1. Multiply unit price by quantity for the exact subtotal.
2. Calculate the raw service fee, then quantize it to cents with `ROUND_HALF_UP`.
3. Add subtotal and fee for the total.
4. Compare the measured and expected masses using an absolute tolerance and no relative tolerance.
5. Use the supplied expression below to turn the Boolean mass fact into display text without introducing a conditional statement:

   PYTHON

   ```
   mass_status = ("FAIL", "PASS")[mass_matches]
   ```

6. Add the converted review-day duration to the aware purchase timestamp.
7. Derive the launch code from the two characters and `strftime("%m%d")`.
8. Combine the important Boolean facts into `record_ready`.

Run assertion group B before formatting the summary.

Stage C: assemble the clearance message

Use adjacent f-strings inside parentheses. Format every Decimal credit value with two decimal places and the review timestamp with `%Y-%m-%d %H:%M UTC`. Preserve the exact line breaks and labels shown in the acceptance example.

5. Run progressive assertions

Group A: cleaned and converted values

PYTHON

```
assert traveler_text == "  aDA BYRON  "
assert (traveler, traveler_key) == ("Ada Byron", "ada byron")
assert item == "Quantum Compass"
assert type(quantity) is int and quantity == 4
assert (unit_price, service_rate) == (Decimal("3.25"), Decimal("0.075"))
assert math.isclose(measured_mass, 0.3, rel_tol=0.0, abs_tol=mass_tolerance)
assert is_priority is True and coupon_code is None
assert purchased_at == datetime.fromisoformat("2026-08-03T14:30:00+00:00")
assert purchased_at.tzinfo is not None
assert review_days == 30
```

Group B: purchase, scanner, and time calculations

PYTHON

```
assert (subtotal, service_fee, total) == (
    Decimal("13.00"),
    Decimal("0.98"),
    Decimal("13.98"),
)
assert mass_matches is True
assert mass_status == "PASS"
assert review_at == datetime.fromisoformat("2026-09-02T14:30:00+00:00")
assert launch_code == "AS0803"
assert record_ready is True
```

Group C: final clearance artifact

PYTHON

```
expected_summary = (
    "=== CHRONONAUT SUPPLY CLEARANCE ===\n"
    "Traveler: Ada Byron [AS0803]\n"
    "Cargo: 4 × Quantum Compass\n"
    "Credits: 13.00 + 0.98 = 13.98\n"
    "Mass check: PASS | Priority: True\n"
    "Review: 2026-09-02 14:30 UTC"
)

assert summary == expected_summary
print(summary)
```

A later pass does not erase an earlier failure

If group B fails because `service_fee` is `0.975`, the Decimal conversion is not the problem. The missing step is the stated cent-quantization rule. Diagnose the first mismatched promise before changing other values. :::

6. Use the hint ladder only when needed

Hint 1

Return to the lesson that owns the blocked boundary:

- Lessons 1 and 5 for conversion, `None`, cleanup, normalization, and slices;
- Lessons 3 and 4 for `math.isclose()`, Decimal construction, and quantization;
- Lesson 7 for explicit yes/no comparison and combined Boolean facts; and
- Lesson 8 for aware ISO parsing and `timedelta` arithmetic.

Run only the nearest assertion group after making one change.

Hint 2

The central calculations have these shapes:

```python
subtotal = unit_price * quantity
service_fee = (subtotal * service_rate).quantize(...)
total = subtotal + service_fee
mass_matches = math.isclose(...)
review_at = purchased_at + timedelta(days=review_days)
```

Supply the named rounding mode and tolerance arguments from the constraints.

Hint 3

The launch code and final readiness can be decomposed as follows:

```python
traveler_letter = traveler[0].upper()
item_letter = item[-1].upper()
date_digits = purchased_at.strftime("%m%d")
launch_code = traveler_letter + item_letter + date_digits

record_ready = (
    quantity > 0
    and mass_matches
    and purchased_at.tzinfo is not None
    and coupon_code is None
)
```

Build the summary one line at a time and compare `repr(summary)` with `repr(expected_summary)` if a hidden space or newline prevents the final pass.

7. Keep debugging evidence

Preserve one failed assertion or exception from your work. Record the exact source value, observed result, expected result, one explanation, the single change you made, and the assertion group that passed afterward.

Failure	Exact evidence	One hypothesis	Controlled change	Verified rerun
What was the first failure?	Include actual and expected values or the final exception line.	Which representation, operation, or format rule explains it?	What one line changed?	Which unchanged group now passes?

Useful puzzle evidence includes:

- `bool(" YES ")` unexpectedly producing `True` for the wrong reason;
- `Decimal(0.075)` carrying a float approximation;
- a service fee left at three decimal places;
- exact float equality rejecting the scanner mass;
- a missing UTC offset; or
- an invisible newline mismatch in the summary.

8. Test the damaged-scanner variation

After the ordinary challenge passes from a clean state, copy your implementation to a new cell and change only:

PYTHON
```
mass_text = "0.42"
```

Recalculate every derived value. For this variation:

PYTHON
```
assert mass_matches is False
assert mass_status == "FAIL"
assert record_ready is False
```

The purchase totals and review timestamp should remain unchanged. Explain why the mass display and readiness fact change while Decimal arithmetic and calendar arithmetic do not.

9. Compare with a solution path

Reveal after all three hint levels have been used or your own checks pass

PYTHON
```
import math
import unicodedata
from datetime import datetime, timedelta
from decimal import Decimal, ROUND_HALF_UP

traveler_text = "  aDA BYRON  "
item_text = "quantum compass"
quantity_text = "4"
unit_price_text = "3.25"
service_rate_text = "0.075"
mass_text = "0.30000000000000004"
priority_text = " YES "
coupon_text = "   "
purchased_at_text = "2026-08-03T14:30:00+00:00"
review_days_text = "30"

expected_mass = 0.3
mass_tolerance = 0.000001
cent = Decimal("0.01")

traveler = traveler_text.strip().title()
traveler_key = unicodedata.normalize("NFC", traveler_text.strip()).casefold()
item = item_text.strip().title()
quantity = int(quantity_text)
```

PYTHON (CONTINUED)

```
unit_price = Decimal(unit_price_text)
service_rate = Decimal(service_rate_text)
measured_mass = float(mass_text)
is_priority = priority_text.strip().casefold() == "yes"
coupon_code = coupon_text.strip() or None
purchased_at = datetime.fromisoformat(purchased_at_text)
review_days = int(review_days_text)

subtotal = unit_price * quantity
service_fee = (subtotal * service_rate).quantize(
    cent,
    rounding=ROUND_HALF_UP,
)
total = subtotal + service_fee
mass_matches = math.isclose(
    measured_mass,
    expected_mass,
    rel_tol=0.0,
    abs_tol=mass_tolerance,
)
mass_status = ("FAIL", "PASS")[mass_matches]
review_at = purchased_at + timedelta(days=review_days)
launch_code = (
    traveler[0].upper()
    + item[-1].upper()
    + purchased_at.strftime("%m%d")
)
record_ready = (
    quantity > 0
    and mass_matches
    and purchased_at.tzinfo is not None
    and coupon_code is None
)

summary = (
    "=== CHRONONAUT SUPPLY CLEARANCE ===\n"
    f"Traveler: {traveler} [{launch_code}]\n"
    f"Cargo: {quantity} × {item}\n"
    f"Credits: {subtotal:.2f} + {service_fee:.2f} = {total:.2f}\n"
    f"Mass check: {mass_status} | Priority: {is_priority}\n"
    f"Review: {review_at:%Y-%m-%d %H:%M UTC}"
)

print(summary)
```

This path keeps external text visible, performs calculations on typed values, and creates display text only at the end. A different implementation is valid when it preserves the names, constraints, and unchanged assertions.

10. Check your understanding

Chrononaut supply clearance checkpoint

Connect each answer to the behavior of your completed artifact.

1. Why is `unit_price` constructed from `unit_price_text` rather than from a float?

- **A.** The text preserves the stated base-ten digits before Decimal construction.
- **B.** Floats cannot be positive.
- **C.** Text automatically applies a currency symbol.
- **D.** Decimal accepts no arithmetic.

2. Why does the scanner check use `math.isclose()`?

- **A.** It converts the mass to text.
- **B.** The measured float may be acceptably close without being exactly the same stored value.
- **C.** It rounds the Decimal price.
- **D.** It adds the review duration.

3. Which evidence shows that `purchased_at` retains timezone meaning?

- **A.** Its year is 2026.
- **B.** Its text contains a hyphen.
- **C.** `purchased_at.tzinfo is not None`
- **D.** The quantity is positive.

4. Why is `summary` assembled after the numeric and datetime checks?

- **A.** Strings perform more exact arithmetic.
- **B.** Assertions cannot inspect numbers.
- **C.** Formatting mutates every source value.
- **D.** Calculation stays in typed values, while the final boundary turns results into display text.

Write your choices before checking the answer key at the end of this book.

11. Decide whether the challenge is complete

Evidence rubric

Evidence	Ready to record when
Typed boundaries	Raw text remains unchanged and every converted value has the promised type and meaning.
Numeric rules	Decimal totals, float tolerance, and rounding stage match the unchanged assertions.
Text and truth	Cleanup, normalization, `None`, priority parsing, code derivation, and readiness facts can be explained.
Time	The parsed timestamp is aware and duration arithmetic produces the expected review instant.
Artifact	The clearance message matches exactly from a clean top-to-bottom run.
Boundary	The damaged-scanner variation changes only the relevant facts.
Debugging	One record connects a real failure to evidence, one hypothesis, one change, and a verified rerun.

This control stores a self-reported marker in this browser. It does not submit work, verify identity, grade the artifact, or issue a certificate.

Key points

- A playful artifact can still require precise representation choices.
- Preserving raw text makes every conversion and cleanup boundary inspectable.
- Decimal money, float measurements, Boolean facts, missing values, and aware timestamps each need their own rules.
- Progressive assertions reveal the first broken promise without exposing the whole solution.
- A boundary variation tests whether one changed input affects only the facts that depend on it.
- The final message is evidence of earlier typed calculations, not a replacement for them.

Milestone Challenge: Pack a Moon-Garden Supply Box

Online lesson and runnable notebook

Write and explain a dispatch-card program using names, integer arithmetic, conversion, and formatting.

- **Goal:** Write and explain a dispatch-card program using names, integer arithmetic, conversion, and formatting.
- **Estimated time:** 2–4 hours of active work.
- **Before starting:** Complete the milestone units and their unit challenges.
- **Keep:** Your source, observed outputs, checks, and a debugging note.

This community challenge is ungraded. Use the checks to review your own work; there is no submission service or automatic grade. Completion is self-reported in this browser on this device.

1. Pack a moon-garden supply box

The box holds 17 seed packets. Each packet costs 125 cents and delivery costs 350 cents per box. Store these as named integers, calculate the subtotal and total, and print the garden name, packet count, and total in dollars with two decimal places. Calculate in cents until display. Use straight-line statements; loops, functions, and collections are not needed yet.

Predict an ordinary result before coding. Start with the smallest part you can run and inspect. Use Colab for notebooks and the local workspace introduced in the units for packaging or command-line tasks.

2. Set up the experiment

PYTHON

```
garden = "Moon Garden"
packets = 17
cents_per_packet = 125
delivery_cents = 350
# Calculate the subtotal and total, then print the dispatch card.
```

The starter is deliberately incomplete. Implement the contract; do not change the expected results to match an incorrect implementation.

3. Check the observable results

1. The packet subtotal is 2,125 cents and the total is 2,475 cents ($24.75).
2. With zero packets the delivery charge still applies: $3.50.
3. With free delivery the original order costs $21.25.
4. Restart the notebook and run from the top without relying on earlier experiments.

Record actual outputs beside predictions. Reduce a failure to a small example, change one cause at a time, and repeat the failed check after repairing it.

4. Change one condition

A helper provides a delivery price as the text "425". Convert it explicitly, explain the types before and after conversion, and recalculate the original order: $25.50. Do not add validation or exception handling yet.

5. Review a hint after attempting the task

Hint: connect the relevant tools

Multiply packets by cents_per_packet, then add delivery_cents. Divide by 100 only for display and use a two-decimal format specification.

6. Explain and self-review

Keep a hand calculation, the printed card, and an annotated source listing. Explain one assignment, expression, and formatting choice. Record one syntax error you repaired and how you located it.

Review path: compare your result with the contract

Check the ordinary result first, then the boundary cases and changed condition. Explain which rule justifies each result. An implementation that differs from the hint can still satisfy the contract; explain why yours does.

Check the reasoning behind your result

Choose an answer, then review the explanation.

1. Why calculate these prices in integer cents?

- **A.** Whole-cent prices are represented exactly in this calculation.
- **B.** All inputs become automatically valid.
- **C.** Delivery becomes free.
- **D.** Syntax errors become impossible.

Write your choices before checking the answer key at the end of this book.

7. Save your work and record completion

Save the notebook or project somewhere you control. Include one decision, one failed check, its repair, and a result you can reproduce. Record completion after reviewing the checks; you can undo it later.

Return to the milestone overview.

Turn Data into Reusable Decisions

Online lesson and runnable notebook

Combine collections, decisions, loops, functions, and copying to allocate limited supplies.

What you will build

Combine collections, decisions, loops, functions, and copying to allocate limited supplies.

Follow the units in order and complete each unit challenge. Keep examples and debugging notes that help explain your choices. This overview is orientation, not a scored activity.

Unit sequence

1. Organizing and Traversing Collections — 7 lessons and a unit challenge.
2. Making Decisions and Repeating Work — 8 lessons and a unit challenge.
3. Building Reusable Functions — 8 lessons and a unit challenge.
4. Sharing, Changing, and Copying Objects — 4 lessons and a unit challenge.

Prepare and pace your work

Complete the preceding milestone in the course sequence, or review its outcomes if you already have that experience. Use each unit's active-time estimate and pause at lesson boundaries. Plan another 2–4 hours for the milestone challenge, including rerunning checks and explaining your result.

Use Colab for notebook exercises and the workspace introduced in the units for local environment, packaging, and command-line tasks.

Connect the units

Finish with Route Supplies through a Puzzle Village. Combine the milestone skills and check a changed condition. The challenge provides a contract, expected results, a hint, and self-review guidance.

Community learning is ungraded. Completion controls record your own progress only in this browser on this device; they do not issue a verified credential.

Begin the first unit.

Unit 3: Organizing and Traversing Collections

Online lesson and runnable notebook

Learn how lists, tuples, dictionaries, sets, traversal, sorting, and nested shapes keep related Python values useful and understandable.

- **Level:** Python Foundations · Unit 3
- **Estimated time:** 45–75 minutes for orientation and readiness work
- **You will learn:** Choose a collection from the relationships the program must preserve, then follow the seven-lesson route through traversal, sorting, and nested data.
- **Practice in:** Google Colab, JupyterLab, or a local editor

1. One game state needs four collection jobs

Imagine a small exploration game. Its current state contains an ordered route, a fixed map coordinate, a named player record, and a group of unique skills:

```python
route = ["dock", "garden", "observatory", "vault"]
position = (12, 7)
player = {"name": "Nova", "energy": 80, "rank": "scout"}
skills = {"mapping", "translation", "repair"}

print(route[0])
print(position)
print(player["energy"])
print("repair" in skills)
```

All four values hold several other values, but they do not promise the same behavior:

- a **list** keeps a deliberate order and can grow, shrink, or change;
- a **tuple** keeps a fixed sequence of positions;
- a **dictionary** connects unique keys with values; and
- a **set** keeps unique members for fast membership and group comparisons.

The question is not “Which collection is best?” There is no best collection in isolation. The useful question is “Which relationships must this part of the program preserve?”

Say the job before naming the type

“Keep every checkpoint in visit order, including repeats” points toward a list. “Find the player’s energy by the key `energy`” points toward a dictionary. A precise statement of the job usually makes the Python type easier to choose. :::

2. Choose from the question the program asks

Use these decision axes before reaching for familiar syntax:

Program requirement	Collection to consider	Important trade-off
Preserve position and permit edits	`list`	Finding an arbitrary value usually scans the list.
Preserve a small fixed positional record	`tuple`	Its positions cannot be replaced, added, or removed.
Look up a value by a stable unique key	`dict`	Keys must be hashable and communicate real identity.
Remove duplicates or compare membership groups	`set`	It offers no numeric position or display-order promise.
Keep an immutable set value	`frozenset`	It cannot be changed after construction.

Ask six questions about new data:

1. Does **order** carry meaning?
2. Are **duplicates** meaningful or noise?
3. Will the program ask by **position**, by **key**, or only by membership?
4. Must the collection **change in place**?
5. Do several fields belong to one **record**?
6. Is the data **nested**, and can you describe the type at every level?

One artifact can legitimately use several answers. For example, a list of dictionaries preserves record order while each dictionary gives fields names.

3. Your path through this unit

Lesson	Capability you will build	Main lab artifact
1	Lists: Keeping Values in Order — index, slice, change, search, add, remove, copy, and explain method returns.	Adventure backpack
2	Tuples and Unpacking Fixed Records — recognize tuple syntax and unpack fixed or starred shapes.	Treasure-map clues
3	Dictionaries: Finding Values by Key — distinguish missing keys, update mappings, use views, and build records, counters, and groups.	Creature codex
4	Sets: Uniqueness and Group Comparisons — deduplicate, compare groups, and use immutable `frozenset` values.	Expedition skills
5	Visiting Items in a Collection — trace what lists, dictionaries, ranges, `enumerate()`, and `zip()` supply.	Route manifest
6	Sorting Collections Without Losing Meaning — rank values and complete records with an explicit stable policy.	Tournament leaderboard
7	Building and Reading Nested Data — navigate one level at a time and refactor shapes for new questions.	Game-world archive

The Unit Challenge combines those capabilities in a signal-vault puzzle. You will decode an ordered archive without mutating its evidence.

4. What you already know from Unit 2

This unit builds on scalar values rather than replacing them. You will reuse:

- integers for positions, counts, and scores;
- strings for IDs, labels, and the final decoded message;
- `None` for a deliberately missing value;
- comparisons and Boolean values for assertions and membership facts;
- string indexing, slicing, and `"".join(...)`; and
- errors as evidence about a value’s type and the attempted operation.

Collections answer a different layer of the design question. `80` can represent one energy value; `{"energy": 80}` preserves the relationship between the label and that value; a list of such dictionaries preserves several player records in an order.

5. Simple traversal now, full control flow next

You need a small amount of `for`-loop syntax to observe what collections supply:

```python
signals = ["north", "east", "home"]

for signal in signals:
    print(signal)
```

Read this as “for each value supplied by `signals`, bind that value to `signal` and run the indented line.” The loop visits every list item in order.

Unit 3 uses simple traversal to explain collection behavior. Unit 4 develops conditional rules, accumulators, filtering, `break`, `continue`, nested loops, `while` loops, and comprehensions. When a Unit 3 lab supplies a small `if` statement, the collection decision—not the branching syntax—is the work being assessed.

6. Set up a collection laboratory

State changes are easier to understand when one notebook cell has one purpose. Use a five-part layout:

```python
# 1. Untouched source evidence
source_route = ["dock", "garden", "vault"]

# 2. A deliberate working copy
working_route = source_route.copy()

# 3. One operation under investigation
removed_stop = working_route.pop(1)

# 4. Observable facts
print(removed_stop)
print(source_route)
print(working_route)

# 5. Checks that state the promises
assert removed_stop == "garden"
assert source_route == ["dock", "garden", "vault"]
assert working_route == ["dock", "vault"]
```

Do not keep rerunning only the final cell after earlier state has changed. Use **Restart and run all** periodically so the notebook proves that its result can be reproduced from a clean state.

Printed braces are not a data model

Two collection values can display with similar punctuation and still offer very different operations. Check `type(value).__name__`, inspect the construction syntax, and state the required relationship instead of guessing from appearance. :::

7. Choose a realistic pace

Unit 3 is planned for approximately **23–34 hours**, including reading, running examples, completing checkpoints, modifying code, doing the seven labs, and solving the challenge. That range is guidance, not a deadline.

A sustainable rhythm is:

1. complete one numbered section and its modification;
2. stop after a checkpoint and explain one answer aloud;
3. return for the next group of operations;
4. do the final lab without opening its support first; and
5. record one error, its actual cause, and the evidence that confirmed the repair.

If the lab feels like copying rather than deciding, close the solution, change the input data, and rebuild the checks for the new case.

8. Before you continue

Run this readiness task from a clean cell:

```python
route = ["dock", "garden", "observatory"]
position = (12, 7)
pilot = {"name": "Nova", "energy": 80}
skills = {"mapping", "repair", "mapping"}

assert route[1] == "garden"
assert position[0] == 12
assert pilot["energy"] == 80
assert skills == {"mapping", "repair"}

print(f"{pilot['name']} starts at {route[0]} with {len(skills)} unique skills.")
```

Then make four changes:

1. add `"vault"` to the end of `route`;
2. replace the energy value with `95` by using its dictionary key;
3. add one new unique skill; and
4. prove with assertions that `position` and the original three route entries still have the expected values.

You are ready for Lesson 1 when you can point to the exact collection each line changes and explain why the other three values remain unchanged.

Key points

Key points

- Choose a collection from the relationships and operations the program needs.
- Lists preserve editable order, tuples preserve fixed positions, dictionaries connect keys to values, and sets preserve unique membership.

- Real programs combine collection types; nested shape should be deliberate.
- Keep source evidence, working state, observations, and assertions separate.
- Unit 3 owns collection behavior and simple traversal; Unit 4 develops complete control-flow design.

:::

References

- Python tutorial: data structures
- Python standard types: sequence, set, and mapping types
- Python built-in functions used with collections

Lists: Keeping Values in Order

Online lesson and runnable notebook

Build and maintain ordered Python lists while distinguishing positions, slices, mutation, method return values, aliases, and separate outer copies.

- **Level:** Python Foundations · Unit 3
- **Estimated time:** 3.5–5 hours
- **You will learn:** Create, inspect, change, search, combine, and copy lists while explaining order, duplicates, mutation, and method returns.
- **Practice in:** Google Colab, JupyterLab, or a local editor

1. Pack an adventure in a deliberate order

An explorer’s backpack is more than a group of names. The first item is easiest to reach, the last was packed most recently, and two water bottles are genuinely two supplies. A list represents those facts:

```python
backpack = ["map", "water", "rope", "water"]

print(backpack)
print(type(backpack).__name__)
print(len(backpack))
```

Python creates one list containing four string references. Lists preserve insertion order, allow duplicates, and are **mutable**: their contents can change without binding the name to a different list.

An empty list can be written either way:

```python
found_items = []
also_empty = list()

assert found_items == also_empty
```

A list may legally mix types:

```python
raw_observation = ["north gate", 3, True, None]
```

That flexibility is useful for an intentionally positional record, but most lists are easier to use when every item has the same role: a list of stop names, scores, or player records. Use a tuple or dictionary when each position has a different fixed meaning.

Questions this lesson will answer

- How can you read one item without confusing its position with its human count?
- Which operations change the same list, and which create a new one?
- Why does `append()` sometimes produce an unexpected list inside a list?
- What do removal methods return, and how do missing items fail?

- When is a copy separate, and where does a shallow copy stop?

2. Positions begin at zero

Python numbers sequence positions from zero:

```python
backpack = ["map", "water", "rope", "torch"]

print(backpack[0])
print(backpack[1])
print(backpack[3])
```

The output is `map`, `water`, and `torch`. Human item number one is Python index zero. An index is an offset from the beginning, not an ordinal label.

Negative indexes count back from the end:

```python
print(backpack[-1])
print(backpack[-2])
```

`-1` selects the last item and `-2` the item before it. This remains useful when the list length changes.

An absent position raises `IndexError`

```python
short_route = ["dock", "garden"]
print(short_route[2])
```

The list has valid indexes `0` and `1`. Asking for index `2` raises `IndexError: list index out of range`. Read the traceback together with `len(short_route)`: the largest valid positive index is one less than the length.

Membership asks a different question:

```python
route = ["dock", "garden", "vault"]

print("garden" in route)
print("tower" not in route)
```

Indexing asks “what is at this position?” Membership asks “does an equal value occur anywhere?” A list normally answers membership by scanning values until it finds a match or reaches the end.

3. Slices select a new list

A slice uses `start:stop`; the start is included and the stop is excluded:

```python
route = ["dock", "garden", "bridge", "tower", "vault"]

middle = route[1:4]
first_two = route[:2]
from_bridge = route[2:]

print(middle)
```

```
PYTHON (CONTINUED)
print(first_two)
print(from_bridge)
```

The results are new outer lists. `middle` contains positions 1, 2, and 3, but not position 4. Omitting a bound means “from the beginning” or “through the end.”

A third slice part is the step:

```
PYTHON
every_other = route[::2]
reversed_copy = route[::-1]

print(every_other)
print(reversed_copy)
print(route)
```

The original `route` remains unchanged. A negative step visits positions in the opposite direction.

Unlike one-item indexing, an oversized slice is safe:

```
PYTHON
print(route[:100])
print(route[100:200])
```

The first expression returns every available item; the second returns an empty list. Slices describe a range of available positions rather than demanding one exact position.

Checkpoint: positions and slices

Positions and slices checkpoint

Choose the result guaranteed by Python’s list rules.

1. For `items = ['map', 'rope', 'torch']`, what is `items[-1]`?

- **A.** `'torch'`
- **B.** `'map'`
- **C.** `-1`
- **D.** An empty list

2. What does `items[1:3]` select from a longer list?

- **A.** Only index 3
- **B.** Indexes 1 and 2
- **C.** Indexes 1, 2, and 3
- **D.** Every item except indexes 1 and 3

3. What happens when `items[:100]` extends beyond the end of a three-item list?

- **A.** It pads the result with `None`.
- **B.** It raises `IndexError`.
- **C.** It returns the available items in a new list.

- **D.** It changes the original list to length 100.

4. Which expression creates a reversed outer copy without changing `items`?

- **A.** `items[-1]`
- **B.** `items.reverse()`
- **C.** `items[1:-1]`
- **D.** `items[::-1]`

Write your choices before checking the answer key at the end of this book.

4. Replace one item or an entire slice

Assigning through an index mutates the existing list:

```python
backpack = ["map", "water", "rope"]
backpack[1] = "canteen"

print(backpack)
```

Only position 1 changes. The list still has three items.

Slice assignment can replace, grow, or shrink a region:

```python
supplies = ["map", "water", "rope", "torch"]

supplies[1:3] = ["canteen", "cord"]
print(supplies)

supplies[1:2] = ["snack", "compass", "blanket"]
print(supplies)

supplies[1:4] = ["provisions"]
print(supplies)
```

The values on the right are spliced into the selected region. Because a list can change length, the replacement does not need the same number of items.

One surprising use inserts without removing anything:

```python
route = ["dock", "vault"]
route[1:1] = ["garden", "tower"]
print(route)
```

The empty slice at position 1 becomes the insertion point.

5. Add one object or add its contents

The shape of the right-hand value determines which operation expresses the job.

`append()` adds exactly one object

```python
backpack = ["map", "rope"]
backpack.append("torch")

print(backpack)
```

The string becomes one new final item. If the argument is itself a list, that list remains one object:

```python
backpack = ["map", "rope"]
emergency_kit = ["bandage", "whistle"]

backpack.append(emergency_kit)
print(backpack)
print(len(backpack))
```

The length is three; position 2 is the entire nested kit.

`extend()` adds values supplied by another iterable

```python
backpack = ["map", "rope"]
emergency_kit = ["bandage", "whistle"]

backpack.extend(emergency_kit)
print(backpack)
```

Now the list is flat and has four strings. Be careful when extending with a string: strings supply characters one at a time.

```python
letters = ["N"]
letters.extend("OVA")
print(letters)
```

This produces `['N', 'O', 'V', 'A']`. Use `append("OVA")` if the complete word should be one item.

`insert()` chooses a position

```python
route = ["dock", "tower", "vault"]
route.insert(1, "garden")
print(route)
```

The former position 1 and everything after it move right. Inserting repeatedly near the beginning requires shifting existing positions; frequent front insertion may suggest a different design later.

`+` creates a new list

```python
day_route = ["dock", "garden"]
night_route = ["tower", "vault"]
complete_route = day_route + night_route

print(day_route)
print(night_route)
print(complete_route)
```

Neither input changes. Concatenation is useful when the program needs to preserve both sources and produce a third sequence.

6. Remove by value or by position

Python offers several removal operations because they answer different questions.

```python
items = ["map", "water", "rope", "water"]

items.remove("water")
print(items)

removed = items.pop()
print(removed)
print(items)
```

`remove(value)` deletes the **first equal value** and returns `None`. `pop()` removes and returns the final item by default.

Choose a position for `pop(index)`:

```python
route = ["dock", "garden", "tower", "vault"]
skipped_stop = route.pop(1)

print(skipped_stop)
print(route)
```

Use `del` when no removed value is needed:

```python
route = ["dock", "garden", "tower", "vault"]
del route[1]
del route[1:2]
print(route)
```

`clear()` removes every item but keeps the same list object:

```python
temporary_clues = ["rune", "key", "star"]
result = temporary_clues.clear()

print(temporary_clues)
print(result)
```

The result is `None`. Methods that mutate a list generally communicate their effect through the changed list, not by returning that list.

Missing removals expose different boundaries

```python
items = ["map", "rope"]
items.remove("torch")
```

This raises `ValueError` because the requested equal value is absent.

```python
items = ["map", "rope"]
items.pop(5)
```

This raises `IndexError` because the position is absent. Decide whether the task identifies an item by equality or by position before choosing a method.

7. Search and count duplicates deliberately

`index()` returns the first matching position:

```python
items = ["map", "water", "rope", "water"]

print(items.index("water"))
print(items.count("water"))
print(items.count("torch"))
```

The outputs are `1`, `2`, and `0`. `count()` has a natural zero result for an absent value; `index()` raises `ValueError` when no match exists.

You can constrain the `index()` search with start and stop positions:

```python
items = ["map", "water", "rope", "water"]
second_water = items.index("water", 2)
print(second_water)
```

This returns index 3 because the search begins at position 2. It still finds only the first match within the chosen range.

Operation cost as a practical choice

Reading `items[5]` uses a known position directly. Checking `target in items`, calling `index()`, `count()`, or `remove()` may inspect many items. That is often fine for a short sequence. If a program repeatedly looks up thousands of values by stable IDs, Lesson 3’s dictionary may fit the question better. :::

Checkpoint: mutation and return values

Mutation and return values checkpoint

Track both the list state and the expression’s result.

1. What does `items.append(['rope', 'torch'])` add to `items`?

- **A.** One nested list object
- **B.** Two separate strings
- **C.** Only the first string
- **D.** Nothing, because lists cannot contain lists

2. After `removed = items.pop(1)`, what does `removed` reference?

- **A.** The changed list
- **B.** The item formerly at index 1
- **C.** The number of remaining items

- **D.** Always `None`

3. What does `['a', 'b', 'a'].remove('a')` do to the list?

- **A.** It removes both occurrences.
- **B.** It leaves the list unchanged.
- **C.** It removes the first equal occurrence.
- **D.** It removes the final item regardless of value.

4. Why is `items = items.append('map')` usually a bug?

- **A.** append() sorts first.
- **B.** append() creates a tuple.
- **C.** Assignment cannot use method calls.
- **D.** append() changes the list and returns `None`, which is then assigned to `items`.

Write your choices before checking the answer key at the end of this book.

8. An alias is not a backup

Assignment does not copy a list:

```python
backpack = ["map", "rope"]
backup = backpack

backup.append("torch")

print(backpack)
print(backup)
```

Both names show the added torch because both names reach the same list object.

The diagram distinguishes a second name for the same list from a separate outer list copy.



Create a separate outer list with `.copy()`, `list(...)`, or a full slice:

```python
backpack = ["map", "rope"]

copy_method = backpack.copy()
copy_constructor = list(backpack)
copy_slice = backpack[:]
```

PYTHON (CONTINUED)

```
copy_method.append("torch")

assert backpack == ["map", "rope"]
assert copy_method == ["map", "rope", "torch"]
assert copy_constructor == backpack
assert copy_slice == backpack
```

Equality tells you the values currently compare the same; changing one outer list proves whether its top-level structure is separate.

An outer copy does not duplicate nested values

PYTHON

```
packs = [["map"], ["rope"]]
outer_copy = packs.copy()

outer_copy[0].append("torch")

print(packs)
print(outer_copy)
```

Both displays contain the torch inside the first nested list. The outer lists are separate, but their positions still refer to the same nested list objects. This is a **shallow copy**. Unit 6 develops the complete identity and nested-copy model; for now, state whether your task needs only a separate outer sequence or fully independent nested state.

9. Visit items without changing the list

A straightforward `for` loop receives list values in order:

PYTHON

```
route = ["dock", "garden", "vault"]

for stop in route:
    print(f"Scanning {stop}")

assert route == ["dock", "garden", "vault"]
```

The target name `stop` is rebound for each iteration. Assigning a new value to that name does not replace the list position:

PYTHON

```
items = ["map", "rope"]

for item in items:
    item = item.upper()

print(items)
```

`items` is unchanged. You would need to assign through a position or build a new list to preserve transformed strings. Lesson 5 examines positions with `enumerate()` and the risks of mutating while traversing. Unit 4 develops filtering and accumulation patterns.

10. Build the explorer backpack

Complete this lab without opening the support first. The source manifest is evidence and must remain unchanged.

```python
source_manifest = ["map", "water", "rope", "water", "torch"]

working_pack = None
first_item = None
last_two = None
water_count = None
removed_item = None
emergency_kit = ["bandage", "whistle"]
final_pack = None
```

Your program must:

1. create a separate outer working list;
2. replace the second item with `"canteen"`;
3. insert `"compass"` immediately after `"map"`;
4. remove and preserve the rope by its position;
5. add both emergency-kit values as separate items;
6. add `"sealed rations"` as one final item;
7. derive the first item, final two items, and original water count; and
8. preserve `source_manifest` and `emergency_kit` exactly.

Run these checks:

```python
assert source_manifest == ["map", "water", "rope", "water", "torch"]
assert emergency_kit == ["bandage", "whistle"]
assert working_pack is not source_manifest
assert removed_item == "rope"
assert first_item == "map"
assert water_count == 2
assert last_two == ["whistle", "sealed rations"]
assert final_pack == [
    "map",
    "compass",
    "canteen",
    "water",
    "torch",
    "bandage",
    "whistle",
    "sealed rations",
]
assert working_pack == final_pack
```

Then modify the source so it begins with two maps and ends with no torch. Update only the assertions whose promised facts genuinely change.

Hint: choose an operation from the shape of each requirement

Use `.copy()` before mutation. Assignment through index replaces one item; `.insert()` chooses a position; `.pop(index)` preserves the removed item; `.extend()` adds the kit's contents; and `.append()` adds the final string as one item. Calculate facts from the appropriate source before changing it.

Show one complete solution after attempting the lab

```python
source_manifest = ["map", "water", "rope", "water", "torch"]
emergency_kit = ["bandage", "whistle"]

first_item = source_manifest[0]
water_count = source_manifest.count("water")

working_pack = source_manifest.copy()
working_pack[1] = "canteen"
working_pack.insert(1, "compass")
removed_item = working_pack.pop(3)
working_pack.extend(emergency_kit)
working_pack.append("sealed rations")

last_two = working_pack[-2:]
final_pack = working_pack.copy()
```

The solution takes facts about the untouched manifest before mutation. The final copy is not required for the mechanics, but it makes the delivered artifact explicit and keeps later edits to `working_pack` from changing its outer list.

Checkpoint: aliases and copy shape

Aliases and copies checkpoint

Decide which list objects and nested values are shared.

1. After `b = a`, what happens when `b.append(4)` if `a` is a list?

- **A.** The same list reached through `a` also contains 4.
- **B.** Python automatically copies `a` first.
- **C.** Only the name `b` changes to 4.
- **D.** append() raises an alias error.

2. Which expression creates a separate outer list with the same current items as `a`?

- **A.** `b = a`
- **B.** `b = a.copy()`
- **C.** `b = a.append([])`
- **D.** `b = a[0]`

3. Why can editing `copy[0]` also affect `original[0]` after `copy = original.copy()`?

- **A.** Lists ignore copy operations.
- **B.** Index zero is always global.
- **C.** Both outer lists can still reference the same nested mutable item.
- **D.** Only tuples can be copied independently.

4. Why does assigning `item = item.upper()` inside `for item in items` not replace list entries?

- **A.** Strings cannot be uppercase.
- **B.** Loops always copy the complete list.

- **C.** The assignment is skipped.
- **D.** It rebinds the loop target name rather than assigning through a list position.

Write your choices before checking the answer key at the end of this book.

11. Explain the choices, not only the result

Before leaving the lesson, answer in your own words:

1. Why does indexing beyond the end fail while an oversized slice succeeds?
2. When should another list be passed to `append()` rather than `extend()`?
3. Which removal operation should you use when the removed value is needed?
4. Why does assigning the result of `append()` destroy the useful list name?
5. What independence does a shallow list copy guarantee, and what does it not guarantee?

Key points

Key points

- Lists preserve order, allow duplicates, and can change length and contents.
- Indexing selects one exact position; slicing returns a new outer list over an available range.
- `append()` adds one object, `extend()` adds supplied items, `insert()` chooses a position, and `+` produces a new list.
- `remove()` deletes the first equal value; `pop()` removes and returns by position; `del` removes without returning the item.
- In-place methods such as `append()`, `extend()`, `remove()`, and `clear()` return `None`.
- Assignment can create an alias. `.copy()`, `list(...)`, and `[:]` create a separate outer list but do not recursively copy nested objects. :::

References

- Python tutorial: more on lists
- Python standard types: common sequence operations
- Python standard types: mutable sequence operations
- Python built-in function: `list`

Tuples and Unpacking Fixed Records

Online lesson and runnable notebook

Represent fixed positional records with tuples, unpack their values into meaningful names, and diagnose comma, immutability, length, and starred-shape boundaries.

- **Level:** Python Foundations · Unit 3
- **Estimated time:** 2.5–4 hours
- **You will learn:** Use tuples for ordered fixed-shape values and unpack ordinary or starred shapes without confusing commas, parentheses, or mutability.
- **Practice in:** Google Colab, JupyterLab, or a local editor

1. Give each clue position one fixed meaning

A treasure scanner produces a row, a column, and a symbol together:

```python
clue = (12, 7, "N")

print(clue)
print(type(clue).__name__)
print(len(clue))
```

This tuple is an ordered three-position record. Position 0 always means row, position 1 column, and position 2 symbol. Unlike an adventure backpack, the clue is not intended to gain or lose positions.

Tuples work well for small values whose **length and positional meaning are fixed**: coordinates, RGB colors, database result rows, or the pair returned by `divmod()`. A dictionary is usually clearer when a record has many optional or self-describing fields.

This lesson answers four practical questions:

- What syntax creates zero-, one-, and many-item tuples?
- Which sequence operations still work when positions cannot be changed?
- How does unpacking make positional meaning visible through names?
- When should a variable-length remainder use one starred target?

2. The comma creates the tuple

Parentheses make tuple boundaries easy to see, but the comma is the decisive syntax:

```python
parenthesized = (12, 7)
packed = 12, 7

assert parenthesized == packed
assert type(packed) is tuple
```

This difference matters for a one-item tuple:

```python
not_a_tuple = (5)
one_item_tuple = (5,)

print(type(not_a_tuple).__name__)
print(type(one_item_tuple).__name__)
```

`(5)` merely groups the integer expression. `(5,)` is a tuple containing one integer. A trailing comma is also accepted in longer tuples and makes multi-line editing safer:

```python
checkpoint = (
    "garden",
    12,
    True,
)
```

The empty tuple needs no comma:

```python
empty = ()
also_empty = tuple()

assert empty == also_empty
```

The `tuple()` constructor consumes values from another iterable:

```python
letters = tuple("NOVA")
numbers = tuple([3, 1, 4])

print(letters)
print(numbers)
```

The string supplies individual characters; the list supplies its integer items.

3. Read a tuple like any ordered sequence

Tuples support length, indexing, negative indexing, slicing, membership, counting, and first-match search:

```python
clue = (12, 7, "N", "N")

print(len(clue))
print(clue[0])
print(clue[-1])
print(clue[1:3])
print("N" in clue)
print(clue.count("N"))
print(clue.index("N"))
```

`clue[1:3]` is a new tuple because slicing preserves the sequence family. The count is two, and `index("N")` returns only the first matching position, index 2.

As with a list, an absent numeric position raises `IndexError` and a missing value passed to `.index()` raises `ValueError`.

Tuple positions cannot be replaced

```python
coordinate = (12, 7)
coordinate[0] = 13
```

Python raises `TypeError: 'tuple' object does not support item assignment`. Tuples also have no `.append()`, `.remove()`, or `.clear()` methods. To represent a different coordinate, create a different tuple:

```python
coordinate = (12, 7)
moved_coordinate = (coordinate[0] + 1, coordinate[1])

print(coordinate)
print(moved_coordinate)
```

The original remains `(12, 7)`.

An immutable tuple can point to a mutable value

```python
mission = ("NOVA", ["map", "rope"])
mission[1].append("torch")

print(mission)
```

The tuple still has the same two positions and still refers to the same list in position 1. The nested list itself permits mutation. “A tuple is immutable” means its own sequence of references cannot be replaced, added, or removed; it does not promise that every reachable object is frozen. Unit 6 explores that object graph in depth.

Checkpoint: tuple syntax and immutability

Tuple syntax and immutability checkpoint

Identify the tuple and the boundary of its immutability.

1. Which expression creates a tuple containing only integer 5?

- **A.** `(5,)`
- **B.** `(5)`
- **C.** `[5]`
- **D.** `{5}`

2. What type does `(10, 20, 30)[1:]` produce?

- **A.** `list`
- **B.** `tuple`
- **C.** `int`
- **D.** `range`

3. What happens when code evaluates `point[0] = 9` for a tuple?

- **A.** The tuple grows.

- **B.** A list is created automatically.
- **C.** Python raises `TypeError` because tuple positions do not support assignment.
- **D.** Only the displayed value changes.

4. Why can `(name, items)` appear to change after `items.append('key')`?

- **A.** Appending replaces the tuple.
- **B.** Tuples become lists when named.
- **C.** The comma stops enforcing types.
- **D.** The tuple still refers to the same mutable list, whose contents changed.

Write your choices before checking the answer key at the end of this book.

4. Unpack fixed positions into meaningful names

Unpacking assigns each supplied position to a target name:

```python
clue = (12, 7, "N")
row, column, symbol = clue

print(row)
print(column)
print(symbol)
```

Read the assignment in two stages. Python evaluates the right side first and obtains three values. It then assigns the first to `row`, the second to `column`, and the third to `symbol`.

The names expose meaning that `clue[0]`, `clue[1]`, and `clue[2]` hide:

```python
is_expected_clue = row == 12 and column == 7 and symbol == "N"
print(is_expected_clue)
```

Parentheses around targets are optional:

```python
(red, green, blue) = (40, 80, 120)
```

Use them when they clarify a nested record boundary, not because unpacking requires them.

Target count must match value count

```python
row, column = (12, 7, "N")
```

Python raises `ValueError: too many values to unpack (expected 2)`.

```python
row, column, symbol = (12, 7)
```

This raises `ValueError: not enough values to unpack`. These errors are evidence that the supplied record shape disagrees with the assignment's promised shape. Do not silence them by adding meaningless throwaway names until you understand which data contract changed.

Swap names without a temporary variable

```python
left = "map"
right = "key"

left, right = right, left

print(left)
print(right)
```

Python first packs the right-side values, then unpacks them into the left-side names. `left` becomes `"key"` and `right` becomes `"map"` without losing either original value.

Unpack values returned together

`divmod()` returns quotient and remainder as a two-item tuple:

```python
complete_crates, leftover_items = divmod(29, 6)

assert complete_crates == 4
assert leftover_items == 5
```

Naming the two meanings immediately is clearer than retaining a value such as `result[0]` and `result[1]`.

5. One starred target absorbs a variable-length remainder

When the first and last positions have fixed meanings but the middle varies, use one starred target:

```python
first, *middle, last = ("N", "O", "V", "A")

print(first)
print(middle)
print(last)
print(type(middle).__name__)
```

`first` is `"N"`, `last` is `"A"`, and `middle` is the **list** `["O", "V"]`. The starred target always collects its share into a list, even when the source is a tuple.

It can appear at the beginning or end:

```python
*earlier, latest = (4, 7, 9, 12)
leader, *followers = ("Nova", "Mira", "Sol")

print(earlier)
print(latest)
print(leader)
print(followers)
```

One assignment may contain only one starred target because two stars would make the division ambiguous:

```text
*beginning, *ending = (1, 2, 3, 4)
```

The syntax is invalid before the program runs.

An empty remainder is valid:

```python
first, *middle, last = ("N", "A")
assert middle == []
```

There are still enough values for the two unstarred targets.

Checkpoint: ordinary and starred unpacking

Unpacking shapes checkpoint

Match target shapes to the values Python supplies.

1. After `row, column = (4, 9)`, what is `row`?

- **A.** `4`
- **B.** `(4, 9)`
- **C.** `9`
- **D.** `[4]`

2. Why does `a, b = (1, 2, 3)` raise `ValueError`?

- **A.** Tuples cannot be assigned.
- **B.** Three supplied values cannot fit two ordinary targets.
- **C.** Integers cannot be unpacked from tuples.
- **D.** Parentheses are missing on the left.

3. What is `middle` after `first, *middle, last = ('N', 'O', 'V', 'A')`?

- **A.** `('O', 'V')`
- **B.** `'OV'`
- **C.** `['O', 'V']`
- **D.** `{'O', 'V'}`

4. What happens for `first, *middle, last = ('N', 'A')`?

- **A.** `last` becomes `None`.
- **B.** The tuple gains an empty item.
- **C.** Python raises an unpacking error.
- **D.** `middle` becomes an empty list.

Write your choices before checking the answer key at the end of this book.

6. Choose tuple or list from the promised lifecycle

Both types preserve order, allow duplicates, and support indexes and slices. The difference is what the program promises to do next:

Requirement	Prefer	Reason
Ordered collection that grows or shrinks	`list`	Its sequence structure is mutable.
Small fixed positional record	`tuple`	Its length and positions cannot be changed.
Named record fields	`dict`	Keys communicate meaning better than distant numeric positions.
Unique membership with no positions	`set`	Uniqueness and group comparisons are its purpose.

Do not choose a tuple merely to prevent every possible change. A tuple containing mutable objects is not deeply frozen, and a value often needs validation beyond its container type. Choose it when fixed positional shape communicates the model.

Hashability is a practical preview

Some tuples can be dictionary keys or set members:

```python
visited_coordinates = {(12, 7), (5, 3)}
terrain_by_coordinate = {(12, 7): "vault", (5, 3): "garden"}

print((12, 7) in visited_coordinates)
print(terrain_by_coordinate[(5, 3)])
```

A tuple is hashable only when all values needed for equality are themselves hashable. A tuple containing a list is not:

```python
invalid_key = ("NOVA", [12, 7])
catalog = {invalid_key: "signal"}
```

This raises `TypeError: unhashable type: 'list'`. Lessons 3 and 4 use the rule; Unit 6 explains hashability systematically.

7. Decode the treasure-map clues

The scanner supplies fixed clue records. Preserve the source tuple and derive named facts and a final word.

```python
source_clues = (
    (12, 7, "N"),
    (4, 11, "O"),
    (9, 2, "V"),
    (1, 5, "A"),
)

first_clue = None
first_row = None
first_column = None
first_symbol = None
remaining_clues = None
second_row = None
second_column = None
```

PYTHON (CONTINUED)

```
second_symbol = None
letters = None
vault_word = None
```

Complete these stages:

1. unpack the first clue and collect all remaining clues with a starred target;
2. unpack the first clue into its three meaningful names;
3. unpack the second clue from `remaining_clues`;
4. create a tuple containing the four symbol positions; and
5. join those symbols into the vault word.

Use these progressive checks:

PYTHON

```
assert source_clues[0] == (12, 7, "N")
assert source_clues[-1] == (1, 5, "A")
assert first_clue == (12, 7, "N")
assert remaining_clues == [
    (4, 11, "O"),
    (9, 2, "V"),
    (1, 5, "A"),
]
assert (first_row, first_column, first_symbol) == (12, 7, "N")
assert (second_row, second_column, second_symbol) == (4, 11, "O")
assert letters == ("N", "O", "V", "A")
assert vault_word == "NOVA"
assert source_clues == (
    (12, 7, "N"),
    (4, 11, "O"),
    (9, 2, "V"),
    (1, 5, "A"),
)
```

For a boundary test, temporarily change one clue to `(4, 11)` and run the line that unpacks it into three names. Record the error message, restore the source, and explain why the failure is useful contract evidence.

Hint: reveal fixed meanings before assembling the word

Use `first_clue, *remaining_clues = source_clues`. Ordinary three-name unpacking then exposes each clue. The final tuple can select index 2 from each of the four known fixed records, after which `"".join(letters)` produces text.

Show one complete solution after attempting the lab

PYTHON

```
source_clues = (
    (12, 7, "N"),
    (4, 11, "O"),
    (9, 2, "V"),
    (1, 5, "A"),
)

first_clue, *remaining_clues = source_clues
first_row, first_column, first_symbol = first_clue
second_row, second_column, second_symbol = remaining_clues[0]

letters = (
```

PYTHON (CONTINUED)

```
    first_symbol,
    second_symbol,
    source_clues[2][2],
    source_clues[3][2],
)
vault_word = "".join(letters)
```

The final two numeric chains are acceptable here because every clue has already been established as the same compact three-position shape. In a larger or less stable record, named dictionary fields would communicate the access better.

Checkpoint: choosing and using fixed records

Fixed-record decisions checkpoint

Choose a representation from the record’s promised shape.

1. Which value best communicates one fixed row-column coordinate?

- **A.** `(12, 7)`
- **B.** `{12, 7}`
- **C.** `[12, 7, 12, 7]`
- **D.** `{'12:7'}`

2. Which type better fits a route that learners will append to and remove stops from?

- **A.** A tuple
- **B.** A list
- **C.** A frozenset
- **D.** An integer

3. Which tuple can be used as a dictionary key?

- **A.** `('zone', ['N', 'O'])`
- **B.** `([1], [2])`
- **C.** `('zone', 3)`
- **D.** `({'row': 1}, 2)`

4. What does an unpacking `ValueError` most directly reveal?

- **A.** Every source value has the wrong type.
- **B.** Tuples are no longer supported.
- **C.** The target names are misspelled.
- **D.** The number of supplied values disagrees with the assignment shape.

Write your choices before checking the answer key at the end of this book.

8. Explain the record contract

Answer these before moving to dictionaries:

1. Why does `(5)` produce an integer while `(5,)` produces a tuple?
2. What part of `("NOVA", ["map"])` is protected from change, and what part is not?
3. Why can unpacking be safer to read than repeated numeric indexing?
4. What type does a starred target produce, and why might that matter later?
5. When would named dictionary fields communicate a record better than tuple positions?

Key points

Key points

- The comma creates a tuple; parentheses normally clarify its boundary.
- Tuples preserve order and duplicates but do not permit replacing, inserting, or deleting their own positions.
- A tuple may still contain a mutable object.
- Ordinary unpacking requires matching value and target counts; one starred target absorbs a variable-length remainder into a list.
- Fixed, compact positional records often suit tuples; changing sequences suit lists, and named records often suit dictionaries.
- A tuple can be a dictionary key or set member only when its relevant contents are hashable. :::

References

- Python tutorial: tuples and sequences
- Python standard types: common sequence operations
- Python language reference: assignment statements and unpacking
- Python built-in function: `tuple`

Dictionaries: Finding Values by Key

Online lesson and runnable notebook

Build keyed catalogs, distinguish missing keys from stored values, update and merge mappings, use dynamic views, and create record, counter, grouping, and lookup shapes.

- **Level:** Python Foundations · Unit 3
- **Estimated time:** 3.5–5 hours
- **You will learn:** Create, inspect, update, merge, remove, traverse, copy, count, and group dictionary values without confusing keys, values, missing data, or source state.
- **Practice in:** Google Colab, JupyterLab, or a local editor

1. Find a creature by ID rather than by position

A field guide should answer “What do we know about creature `mossling`?” A list could store creature records, but repeated lookup would require finding the right position. A dictionary connects a stable **key** directly to a **value**:

```python
creature_energy = {
    "mossling": 30,
    "skyfin": 55,
    "emberfox": 42,
}

print(creature_energy["skyfin"])
print(len(creature_energy))
print(type(creature_energy).__name__)
```

The key `"skyfin"` maps to integer value `55`. Keys are unique within one dictionary. Values may repeat.

Python dictionaries preserve **insertion order**, but their central purpose is lookup by key, not access by numeric position. If visible order is the only requirement, a list may communicate that job more directly.

An empty pair of braces is a dictionary, not a set:

```python
empty_catalog = {}
also_empty = dict()

assert type(empty_catalog) is dict
assert empty_catalog == also_empty
```

This lesson will help you answer:

- Is a key absent, or is its stored value deliberately `None`?
- Does an operation mutate the dictionary or create a new mapping?
- What do keys, values, and item views supply during traversal?

- When does a dictionary represent one record, a lookup table, a counter, or a grouping?

2. Direct lookup makes missing keys visible

Square brackets state that the key must exist:

```python
creature_energy = {"mossling": 30, "skyfin": 55}

print(creature_energy["mossling"])
print("skyfin" in creature_energy)
print("dragon" not in creature_energy)
```

Dictionary membership checks **keys**, not values:

```python
print("mossling" in creature_energy)
print(30 in creature_energy)
print(30 in creature_energy.values())
```

The results are `True`, `False`, and `True`. State the side of the mapping you intend to search.

An absent direct key raises `KeyError`:

```python
print(creature_energy["dragon"])
```

The traceback includes the requested key. That failure is useful when absence means the program's data contract is broken.

`.get()` supports expected absence

```python
creature_energy = {"mossling": 30, "skyfin": 55}

print(creature_energy.get("skyfin"))
print(creature_energy.get("dragon"))
print(creature_energy.get("dragon", 0))
```

`.get(key)` returns `None` for an absent key; `.get(key, default)` returns the provided default. It does not add that key to the dictionary.

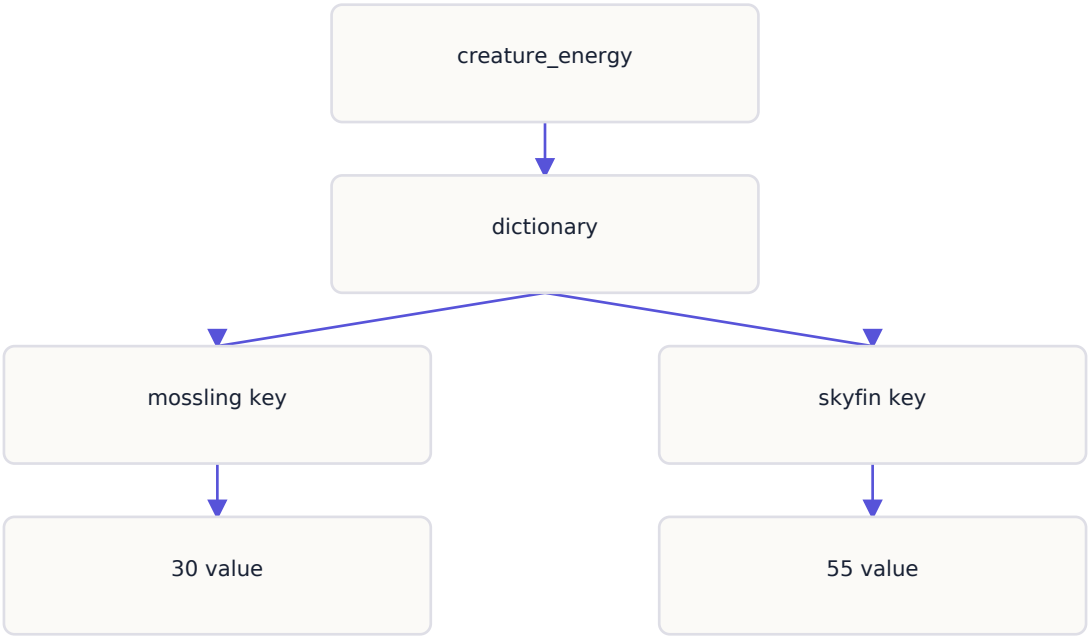
Missing and stored-as-`None` are different states

```python
observations = {
    "mossling": "near the river",
    "skyfin": None,
}

print(observations.get("skyfin"))
print(observations.get("dragon"))
print("skyfin" in observations)
print("dragon" in observations)
```

Both `.get()` calls print `None`, but membership distinguishes the states. The `skyfin` key exists and deliberately stores a missing observation; `dragon` has no entry. Use `key in mapping` when that distinction changes the program's meaning.

A dictionary connects unique lookup keys to values; the name references the complete mapping rather than one pair.



Checkpoint: lookup and missing states

Dictionary lookup checkpoint

Choose the operation that preserves the intended missing-data meaning.

1. What does `'mossling' in catalog` check for a dictionary?

- **A.** Whether `'mossling'` is a key
- **B.** Whether every value equals `'mossling'`
- **C.** Whether `'mossling'` is the first inserted value
- **D.** Whether the dictionary has exactly one pair

2. Which expression makes an absent required key fail with `KeyError`?

- **A.** `catalog.get('dragon')`
- **B.** `catalog['dragon']`
- **C.** `'dragon' in catalog`
- **D.** `catalog.values()`

3. What does `counts.get('north', 0)` return when `'north'` is absent?

- **A.** It raises `IndexError`.
- **B.** It adds `'north': 0` and returns the dictionary.
- **C.** It returns `0` without adding the key.
- **D.** It always returns `None`.

4. How can code distinguish an absent key from a key whose value is `None`?

- **A.** Call `.get()` twice.
- **B.** Compare the dictionary with `None`.
- **C.** Use `.count(None)`.
- **D.** Check `key in mapping` as well as retrieving the value.

Write your choices before checking the answer key at the end of this book.

3. Add and replace pairs with assignment

Assignment through a key adds an absent pair or replaces an existing value:

```
PYTHON
creature_energy = {"mossling": 30, "skyfin": 55}

creature_energy["emberfox"] = 42
creature_energy["mossling"] = 35

print(creature_energy)
```

The dictionary now has three keys. Replacing `mossling` does not move it to the end; its original insertion position remains.

`.update()` applies several pairs in place:

```
PYTHON
catalog = {"mossling": 30, "skyfin": 55}
changes = {"skyfin": 60, "emberfox": 42}

result = catalog.update(changes)

print(catalog)
print(result)
```

The existing `skyfin` value is replaced, `emberfox` is added, and the method returns `None`.

The merge operator creates a new dictionary:

```
PYTHON
original = {"mossling": 30, "skyfin": 55}
changes = {"skyfin": 60, "emberfox": 42}

merged = original | changes

print(original)
print(merged)
```

When both operands contain the same key, the right operand’s value wins. The original mapping remains unchanged.

The in-place merge operator changes its left operand:

```
PYTHON
working_catalog = {"mossling": 30}
working_catalog |= {"skyfin": 55}
print(working_catalog)
```

Choose from the state requirement: use `|` when both inputs must remain available; use update/assignment when the existing working mapping is meant to change.

4. Remove a pair and know what is returned

`pop(key)` removes a requested key and returns its value:

```python
catalog = {"mossling": 30, "skyfin": 55, "emberfox": 42}
retired_energy = catalog.pop("skyfin")

print(retired_energy)
print(catalog)
```

An absent key raises `KeyError`, unless you supply a default:

```python
missing_energy = catalog.pop("dragon", None)
print(missing_energy)
```

No pair is added. Use a default only when absence is an expected state; it can otherwise hide a misspelled key.

`del` removes without returning the value:

```python
catalog = {"mossling": 30, "skyfin": 55}
del catalog["mossling"]
print(catalog)
```

`popitem()` removes and returns the most recently inserted remaining pair:

```python
catalog = {"mossling": 30, "skyfin": 55, "emberfox": 42}
last_pair = catalog.popitem()

print(last_pair)
print(type(last_pair).__name__)
```

The result is the tuple `('emberfox', 42)`. This is useful for last-in processing, not for choosing an arbitrary key. Calling it on an empty dictionary raises `KeyError`.

`clear()` removes all pairs and returns `None`.

5. Views stay connected to the dictionary

`.keys()`, `.values()`, and `.items()` expose different sides of a mapping:

```python
catalog = {"mossling": 30, "skyfin": 55}

keys = catalog.keys()
values = catalog.values()
pairs = catalog.items()

print(keys)
print(values)
print(pairs)
```

These are **view objects**, not frozen list snapshots. If the mapping changes, the views reflect its current state:

```python
keys = catalog.keys()
catalog["emberfox"] = 42

print(keys)
print("emberfox" in keys)
```

Create a list explicitly when you need a separate positional snapshot:

```python
key_snapshot = list(catalog.keys())
catalog["cloudwhale"] = 90

print(key_snapshot)
print(list(catalog.keys()))
```

Direct traversal supplies keys

```python
catalog = {"mossling": 30, "skyfin": 55}

for creature_id in catalog:
    print(creature_id)
```

Use `.values()` when only values matter, or `.items()` to unpack key-value tuples:

```python
for creature_id, energy in catalog.items():
    print(f"{creature_id}: {energy}")
```

Do not write `for key, value in catalog:` and expect pairs. Direct traversal receives each key; a string key might then be incorrectly unpacked character by character or fail from the wrong length.

Checkpoint: updates, removals, and views

Updates, removals, and views checkpoint

Track mutation, return values, duplicate keys, and traversal inputs.

1. What is `({'x': 1} | {'x': 9})['x']`?

- **A.** `9`
- **B.** `1`
- **C.** `[1, 9]`
- **D.** A duplicate-key error

2. What does `removed = catalog.pop('skyfin')` store in `removed`?

- **A.** The changed dictionary
- **B.** The value associated with `'skyfin'`
- **C.** The key string `'skyfin'`
- **D.** Always `None`

3. What happens to a previously created `keys = catalog.keys()` view after a new key is added?

- **A.** It becomes invalid.
- **B.** It remains a frozen two-key list.
- **C.** It reflects the dictionary's current keys.
- **D.** It turns into a set automatically.

4. What values does `for item in catalog:` receive?

- **A.** Only dictionary values
- **B.** Numeric positions
- **C.** Two-item lists
- **D.** Dictionary keys

Write your choices before checking the answer key at the end of this book.

6. Keys must be unique and hashable

Writing a duplicate key in one literal retains the last value:

```python
levels = {"mossling": 1, "mossling": 2}
print(levels)
```

The dictionary contains one pair: `"mossling": 2`. Duplicate key syntax is not a way to store several values; use a list as the one value when grouping is the job.

Common immutable scalar values—strings, integers, and suitable tuples—can be keys:

```python
terrain = {
    (12, 7): "vault",
    (5, 3): "garden",
}

print(terrain[(12, 7)])
```

A list cannot be a key:

```python
invalid_catalog = {[12, 7]: "vault"}
```

Python raises `TypeError: unhashable type: 'list'`. A key must have a stable hash and equality behavior while it participates in the dictionary. Unit 6 develops the full rule; for now, stable IDs and immutable coordinate tuples are safe, readable choices.

Lookup is the dictionary's strength

A dictionary is designed to find a value from its key without scanning every pair in the ordinary case. Searching for an arbitrary value still requires inspecting values unless the program maintains a second reverse lookup. Use this cost intuition to choose a shape; Unit 7 treats algorithmic efficiency more formally. :::

7. One dictionary can represent one named record

Keys make heterogeneous fields readable:

```python
creature = {
    "id": "mossling",
    "energy": 30,
    "habitat": "riverbank",
    "discovered": True,
    "nickname": None,
}

print(creature["habitat"])
creature["energy"] = 35
```

`creature["energy"]` explains its meaning more clearly than `creature[1]` when the record has several fields. Keep field names consistent across similar records; Lesson 7 shows how malformed nested shapes fail.

An optional field may be absent, explicitly `None`, or an empty collection. Those states should be chosen, not accidentally collapsed:

```python
record = {"id": "skyfin", "nickname": None, "sightings": []}

assert "notes" not in record
assert "nickname" in record and record["nickname"] is None
assert "sightings" in record and record["sightings"] == []
```

8. Dictionaries also count and group

For a counter, each distinct item becomes a key and its occurrence count becomes the value:

```python
regions = ["north", "east", "north", "west", "north"]
counts = {}

for region in regions:
    counts[region] = counts.get(region, 0) + 1

print(counts)
```

Trace the first `"north"`: it is absent, so `.get(..., 0)` supplies zero; adding one stores `1`. The later occurrences retrieve and increment the existing value. The loop pattern itself is developed further in Unit 4; here the important model is `{value_being_counted: count}`.

A grouping dictionary maps each category to a list of complete items:

```python
observations = (
    ("north", "mossling"),
    ("east", "skyfin"),
    ("north", "emberfox"),
)
groups = {}

for region, creature_id in observations:
    if region not in groups:
        groups[region] = []
```

PYTHON (CONTINUED)
```
    groups[region].append(creature_id)

print(groups)
```

After the explicit steps make sense, `.setdefault()` can express “retrieve the list, creating it if absent”:

PYTHON
```
groups = {}

for region, creature_id in observations:
    groups.setdefault(region, []).append(creature_id)
```

`.setdefault(key, default)` inserts the default only when the key is absent and then returns the stored value. It is concise, but do not use it before you can explain which list receives `.append()`.

9. A shallow copy separates only the outer mapping

PYTHON
```
source = {
    "name": "mossling",
    "skills": ["hide", "heal"],
}
snapshot = source.copy()

snapshot["name"] = "elder mossling"
snapshot["skills"].append("glow")

print(source)
print(snapshot)
```

The top-level `"name"` fields differ because the outer dictionaries are separate. Both `"skills"` keys still refer to the same nested list, so both displays include `"glow"`. `dict(source)` and `{**source}` also make shallow outer copies. Unit 6 owns the choice between shared nested state, deliberate reconstruction, and deep copying.

Checkpoint: records, counters, and grouping

Dictionary modeling checkpoint

Choose the mapping shape that preserves the required relationship.

1. Which expression reads the energy field from a dictionary record named `creature`?

- **A.** `creature['energy']`
- **B.** `creature.energy()`
- **C.** `creature[energy]` when no such name exists
- **D.** `creature.index('energy')`

2. Which mapping best represents occurrence counts for regions?

- **A.** `{1: 'north', 2: 'east'}`
- **B.** `{'north': 3, 'east': 1}`
- **C.** `{'north', 'east'}`

- **D.** `('north', 3, 'east', 1)`

3. Which shape keeps every creature grouped under its region?

- **A.** `{'north': 2}`
- **B.** `['north', 'mossling']`
- **C.** `{'north': ['mossling', 'emberfox']}`
- **D.** `{'north', 'mossling', 'emberfox'}`

4. After `copy = source.copy()`, why can `copy['skills'].append('glow')` affect `source`?

- **A.** Dictionary keys are always global.
- **B.** copy() returns the same outer dictionary.
- **C.** append() ignores its receiver.
- **D.** Both outer dictionaries may still refer to the same nested skills list.

Write your choices before checking the answer key at the end of this book.

10. Build a creature codex

Preserve the source tuple of records while producing a direct lookup, a working update, habitat counts, and grouped creature IDs.

```python
source_creatures = (
    {"id": "mossling", "energy": 30, "habitat": "river"},
    {"id": "skyfin", "energy": 55, "habitat": "cliffs"},
    {"id": "emberfox", "energy": 42, "habitat": "river"},
)

creatures_by_id = {}
working_mossling = None
habitat_counts = {}
creatures_by_habitat = {}
removed_skyfin = None
```

Complete these stages:

1. traverse the source and map each creature ID to its complete record;
2. create a separate outer dictionary for the mossling record and increase only that working energy to `35`;
3. count records by habitat;
4. group creature IDs into lists by habitat;
5. create a separate working copy of the lookup and `pop()` the skyfin record; store the returned record in `removed_skyfin`; and
6. prove that the source tuple and original lookup still contain their original data.

```python
assert list(creatures_by_id) == ["mossling", "skyfin", "emberfox"]
assert creatures_by_id["skyfin"]["energy"] == 55
assert working_mossling == {
    "id": "mossling",
```

PYTHON (CONTINUED)

```
    "energy": 35,
    "habitat": "river",
}
assert source_creatures[0]["energy"] == 30
assert habitat_counts == {"river": 2, "cliffs": 1}
assert creatures_by_habitat == {
    "river": ["mossling", "emberfox"],
    "cliffs": ["skyfin"],
}
assert removed_skyfin["id"] == "skyfin"
assert "skyfin" in creatures_by_id
assert source_creatures == (
    {"id": "mossling", "energy": 30, "habitat": "river"},
    {"id": "skyfin", "energy": 55, "habitat": "cliffs"},
    {"id": "emberfox", "energy": 42, "habitat": "river"},
)
```

Then add a fourth source record in habitat `"cliffs"`. Rerun cleanly and update the expected order, count, and group without hard-coding a special case for that creature.

Hint: build each dictionary around one lookup question

For the direct lookup, assign `record["id"]` as a key and the complete `record` as its value. Copy the mossling record before replacing its energy. For each habitat, use `.get(habitat, 0) + 1` in the counter and create a list before appending in the grouping dictionary. Pop from a copy of the complete lookup.

Show one complete solution after attempting the lab

PYTHON

```
source_creatures = (
    {"id": "mossling", "energy": 30, "habitat": "river"},
    {"id": "skyfin", "energy": 55, "habitat": "cliffs"},
    {"id": "emberfox", "energy": 42, "habitat": "river"},
)

creatures_by_id = {}
habitat_counts = {}
creatures_by_habitat = {}

for record in source_creatures:
    creature_id = record["id"]
    habitat = record["habitat"]
    creatures_by_id[creature_id] = record
    habitat_counts[habitat] = habitat_counts.get(habitat, 0) + 1
    if habitat not in creatures_by_habitat:
        creatures_by_habitat[habitat] = []
    creatures_by_habitat[habitat].append(creature_id)

working_mossling = creatures_by_id["mossling"].copy()
working_mossling["energy"] = 35

working_lookup = creatures_by_id.copy()
removed_skyfin = working_lookup.pop("skyfin")
```

The lookup values deliberately share the untouched source records. The lab never mutates those nested records. The one edited record is copied first, and the one removed pair is popped from a separate outer lookup.

11. Explain the mapping decisions

1 When is square-bracket lookup better evidence than `.get()`?

2 Why does dictionary membership not answer whether a value occurs?

3 What do `update()` and `|` do differently to their input mappings?

4 Why is a dictionary view different from `list(dictionary)`?

5 Which question is answered by a record, a counter, a grouping, and a direct lookup?

Key points

Key points

- Dictionaries map unique hashable keys to values and preserve insertion order.
- Square brackets require a key; `.get()` supports expected absence; membership distinguishes an absent key from a stored `None`.
- Assignment and `update()` mutate; `|` creates a new merged dictionary; the right-side value wins for duplicate keys.
- `pop()` returns a removed value, `popitem()` returns the latest pair, and `del` removes without returning the value.
- Keys, values, and item views remain connected to the mapping; direct traversal supplies keys.
- Dictionaries can model named records, keyed lookups, counters, and groups.
- `.copy()` creates a separate outer mapping but retains nested value references. :::

References

- Python tutorial: dictionaries
- Python standard types: mapping types and dictionary operations
- Python standard types: dictionary view objects
- Python built-in function: `dict`

Sets: Uniqueness and Group Comparisons

Online lesson and runnable notebook

Use sets for unique membership and group comparisons, choose deliberate mutation methods, and represent immutable hashable groups with frozenset.

- **Level:** Python Foundations · Unit 3
- **Estimated time:** 3–4.5 hours
- **You will learn:** Create, change, compare, and freeze unique-value groups while respecting missing-item, ordering, indexing, and hashability boundaries.
- **Practice in:** Google Colab, JupyterLab, or a local editor

1. Keep one copy of every discovered skill

An expedition log may repeat a skill whenever a crew member demonstrates it. A readiness check cares only whether each skill is available at least once:

```python
skill_log = ["mapping", "repair", "mapping", "translation", "repair"]
available_skills = set(skill_log)

print(len(skill_log))
print(len(available_skills))
print("repair" in available_skills)
```

The set has three members because equal duplicates collapse. A set models **unique membership**, not occurrence count or positional order. Preserve the original list when arrival order or duplicate evidence still matters.

Set literals use braces:

```python
required_skills = {"mapping", "translation", "repair"}
```

An empty set is the important exception:

```python
empty_set = set()
empty_dictionary = {}

print(type(empty_set).__name__)
print(type(empty_dictionary).__name__)
```

`{}` already denotes an empty dictionary, so use `set()` when no members exist.

This lesson answers:

- Which operations mutate a set, and how do missing removals differ?
- How do union, intersection, difference, and symmetric difference answer distinct questions?

- What do subset, superset, and disjoint relationships prove?
- When does an immutable `frozenset` fit where a mutable set cannot?

2. Membership matters; positions do not

Sets support `len()` and membership:

```python
visited = {"dock", "garden", "vault"}

assert len(visited) == 3
assert "garden" in visited
assert "tower" not in visited
```

They do not support numeric indexing:

```python
visited = {"dock", "garden", "vault"}
print(visited[0])
```

Python raises `TypeError: 'set' object is not subscriptable`. There is no promised “first” member for index zero to select.

Printing or traversing a set can show members in an order that looks stable in one session, but code must not treat it as a positional or display-order contract. Create an explicitly sorted list when presentation needs a predictable order:

```python
visited = {"dock", "garden", "vault"}
display_names = sorted(visited)
print(display_names)
```

Lesson 6 develops sorting policies. The result here is a list; the set remains unchanged.

Membership is a natural set question

Sets are designed for repeated membership checks without scanning a positional sequence in the ordinary case. Keep a list as well if you need both original order and membership behavior. One collection does not need to satisfy every requirement. :::

3. Add one member or update from many values

`.add()` treats its argument as one member:

```python
skills = {"mapping", "repair"}
result = skills.add("translation")

print("translation" in skills)
print(result)
```

The set changes and the method returns `None`.

Adding an existing equal member has no effect on length:

```python
before = len(skills)
skills.add("mapping")
after = len(skills)

assert before == after
```

`.update()` consumes members from one or more iterables:

```python
skills = {"mapping"}
skills.update(["repair", "translation"], {"navigation"})

assert skills == {"mapping", "repair", "translation", "navigation"}
```

A string supplies individual characters, just as it did for `list.extend()`:

```python
letters = {"N"}
letters.update("OVA")
assert letters == {"N", "O", "V", "A"}
```

Use `.add("NOVA")` if the complete word should be one member.

4. Choose removal behavior from the missing-item rule

`.remove()` requires the member:

```python
skills = {"mapping", "repair", "translation"}
skills.remove("repair")

assert "repair" not in skills
```

Removing an absent member raises `KeyError`:

```python
skills.remove("flying")
```

`.discard()` makes absence a harmless no-op:

```python
skills = {"mapping", "translation"}
result = skills.discard("flying")

assert skills == {"mapping", "translation"}
assert result is None
```

Use `remove()` when absence violates the task's promise and `discard()` when "make sure it is absent" is the complete requirement.

`.pop()` removes and returns an arbitrary member:

```python
tokens = {"sun", "moon", "star"}
removed_token = tokens.pop()

assert removed_token not in tokens
```

```
PYTHON (CONTINUED)
assert len(tokens) == 2
```

Do not assert which token is removed. Set `pop()` does not mean “remove the last item.” Calling it on an empty set raises `KeyError`.

`.clear()` removes every member and returns `None`.

Checkpoint: creation and mutation

Set creation and mutation checkpoint

Choose the operation that matches the uniqueness and missing-item requirement.

1. Which expression creates an empty set?

- **A.** `set()`
- **B.** `{}`
- **C.** `[]`
- **D.** `()`

2. Which operation ensures `'flying'` is absent without failing when it was already absent?

- **A.** `skills.remove('flying')`
- **B.** `skills.discard('flying')`
- **C.** `skills.pop('flying')`
- **D.** `skills[0] = 'flying'`

3. What does `tokens.pop()` guarantee for a non-empty set?

- **A.** It removes the alphabetically last member.
- **B.** It returns `None` and keeps every member.
- **C.** It removes and returns some member, without a last-position promise.
- **D.** It always removes the first inserted member.

4. What happens when `.add()` receives a value already equal to a set member?

- **A.** The member is stored twice.
- **B.** The set becomes a list.
- **C.** Python raises `ValueError`.
- **D.** The set remains unchanged in membership and size.

Write your choices before checking the answer key at the end of this book.

5. Set operations answer four different group questions

Suppose two crews have these skills:

```python
nova_skills = {"mapping", "repair", "translation"}
mira_skills = {"navigation", "repair", "first aid"}
```

Union: present in either group

```python
combined = nova_skills | mira_skills
same_result = nova_skills.union(mira_skills)

assert combined == same_result
assert combined == {
    "mapping",
    "repair",
    "translation",
    "navigation",
    "first aid",
}
```

Union combines membership and keeps one copy of overlap.

Intersection: present in both groups

```python
shared = nova_skills & mira_skills
assert shared == {"repair"}
```

The named spelling is `nova_skills.intersection(mira_skills)`.

Difference: present only on the left

```python
nova_only = nova_skills - mira_skills
mira_only = mira_skills - nova_skills

assert nova_only == {"mapping", "translation"}
assert mira_only == {"navigation", "first aid"}
```

Difference is directional. Swapping operands changes the question.

Symmetric difference: present in exactly one group

```python
not_shared = nova_skills ^ mira_skills
assert not_shared == nova_only | mira_only
```

The shared `"repair"` member is excluded. The named method is `.symmetric_difference()`.

All four expressions above create new sets; neither source set changes. In-place forms such as `|=`, `&=`, `-=`, and `^=` mutate the left set and should be used only when that state change is intended.

6. Compare readiness with subset relationships

An expedition is ready if all required skills are included in its available skills:

```python
required = {"mapping", "repair"}
available = {"mapping", "repair", "translation"}
```

PYTHON (CONTINUED)

```
print(required <= available)
print(required < available)
print(available >= required)
print(available > required)
```

- `<=` means subset, including equality;
- `<` means proper subset, requiring at least one additional member on the right;
- `>=` means superset, including equality; and
- `>` means proper superset.

Equality ignores insertion history and display order:

PYTHON

```
assert {"map", "rope"} == {"rope", "map"}
```

`.isdisjoint()` asks whether two groups share no members:

PYTHON

```
hazards = {"flood", "storm"}
current_conditions = {"clear", "cold"}

assert hazards.isdisjoint(current_conditions)
```

This expresses “no overlap” more directly than constructing an intersection only to compare it with an empty set.

Checkpoint: group operations and relationships

Set operations and relationships checkpoint

Translate each group question into the matching set operation.

1. Which expression finds skills present in both `a` and `b`?

- **A.** `a & b`
- **B.** `a | b`
- **C.** `a - b`
- **D.** `a ^ b`

2. Which expression finds members of `required` missing from `available`?

- **A.** `available | required`
- **B.** `required - available`
- **C.** `available & required`
- **D.** `required <= available`

3. Which fact proves every required skill appears in `available`?

- **A.** `required > available`
- **B.** `required.isdisjoint(available)`

- **C.** `required <= available`
- **D.** `required ^ available`

4. Which method directly asks whether two sets share no members?

- **A.** `.union()`
- **B.** `.update()`
- **C.** `.issuperset()`
- **D.** `.isdisjoint()`

Write your choices before checking the answer key at the end of this book.

7. Set members must be hashable

Strings, numbers, and suitable tuples can be members:

```python
known_locations = {(12, 7), (4, 9)}
assert (12, 7) in known_locations
```

A list cannot:

```python
invalid_locations = {[12, 7], [4, 9]}
```

Python raises `TypeError: unhashable type: 'list'`. A mutable list could change in a way that invalidates the set's membership organization. A tuple works only if its own relevant contents are hashable.

8. `frozenset` represents an immutable set value

Construct a frozen set from any iterable:

```python
required_skills = frozenset(["mapping", "repair", "translation"])

print(type(required_skills).__name__)
print("repair" in required_skills)
```

It supports non-mutating set operations and comparisons:

```python
available = {"mapping", "repair", "translation", "navigation"}

assert required_skills <= available
assert required_skills & available == required_skills
```

It has no `.add()`, `.remove()`, or `.update()` methods:

```python
required_skills.add("navigation")
```

The resulting `AttributeError` confirms the immutable interface.

Because a `frozenset` is hashable when its members are hashable, it can itself be a set member or dictionary key:

```python
mission_by_requirements = {
    frozenset({"mapping", "repair"}): "river survey",
    frozenset({"translation", "navigation"}): "signal hunt",
}

mission_key = frozenset({"repair", "mapping"})
print(mission_by_requirements[mission_key])
```

Set equality ignores order, so the differently written key finds the same requirements group.

A set can contain frozen sets:

```python
approved_teams = {
    frozenset({"Nova", "Mira"}),
    frozenset({"Sol", "Ivo"}),
}

assert frozenset({"Mira", "Nova"}) in approved_teams
```

Choose `frozenset` because the value is conceptually an immutable membership group or must be hashable—not merely because its name sounds safer.

Checkpoint: hashability and `frozenset`

Hashability and frozenset checkpoint

Choose a representation that remains a valid set member or dictionary key.

1. Which value can be a member of a set?

- **A.** `(12, 7)`
- **B.** `[12, 7]`
- **C.** `{'row': 12}`
- **D.** `{'north', 'east'}`

2. Which statement about `frozenset` is correct?

- **A.** It preserves numeric indexes.
- **B.** It supports membership and set operations but has no mutation methods.
- **C.** It stores duplicate members.
- **D.** It is a sorted list with a different name.

3. Which expression can represent a set whose members are immutable skill groups?

- **A.** `{{'map'}, {'repair'}}`
- **B.** `{['map'], ['repair']}`
- **C.** `{frozenset({'map'}), frozenset({'repair'})}`
- **D.** `set(set('map'))`

4. Why can `frozenset({'a', 'b'})` retrieve a dictionary entry stored under `frozenset({'b', 'a'})`?

- **A.** Dictionaries ignore all keys.
- **B.** Strings are automatically sorted in source code.
- **C.** The first key mutates into the second.
- **D.** Set equality is based on membership, not insertion order.

Write your choices before checking the answer key at the end of this book.

9. Solve the expedition-skills puzzle

Three explorers report skills with duplicates. Build stable evidence and answer the readiness questions without relying on set display order.

PYTHON

```
nova_log = ["mapping", "repair", "mapping", "translation"]
mira_log = ["navigation", "repair", "first aid", "repair"]

required = frozenset({"mapping", "repair", "translation"})
nova_skills = None
mira_skills = None
shared_skills = None
all_skills = None
nova_missing = None
mira_missing = None
nova_ready = None
mira_ready = None
team_signature = None
display_skills = None
```

Your artifact must:

1. convert each log to a unique skill set without changing either log;
2. find shared and combined membership;
3. find required skills missing from each explorer;
4. calculate readiness with a subset relationship;
5. create an immutable `team_signature` from the combined skills; and
6. create a predictable alphabetical list only for display.

PYTHON

```
assert nova_log == ["mapping", "repair", "mapping", "translation"]
assert mira_log == ["navigation", "repair", "first aid", "repair"]
assert nova_skills == {"mapping", "repair", "translation"}
assert mira_skills == {"navigation", "repair", "first aid"}
assert shared_skills == {"repair"}
assert all_skills == {
    "mapping",
    "repair",
    "translation",
    "navigation",
    "first aid",
}
assert nova_missing == set()
assert mira_missing == {"mapping", "translation"}
assert nova_ready is True
assert mira_ready is False
```

PYTHON (CONTINUED)

```
assert type(team_signature) is frozenset
assert team_signature == all_skills
assert display_skills == [
    "first aid",
    "mapping",
    "navigation",
    "repair",
    "translation",
]
```

Boundary variation: add two more `"repair"` entries and one `"navigation"` entry to `nova_log`. Predict which assertions change before rerunning. The unique set should gain only navigation; duplicate repair records remain preserved in the log but not the set.

Hint: translate each sentence into one group question

Convert logs with `set(...)`. Shared means intersection, combined means union, missing means `required - available`, and ready means `required <= available`. Pass the combined set to `frozenset()` and to `sorted()` for two different final representations.

Show one complete solution after attempting the puzzle

PYTHON

```
nova_log = ["mapping", "repair", "mapping", "translation"]
mira_log = ["navigation", "repair", "first aid", "repair"]
required = frozenset({"mapping", "repair", "translation"})

nova_skills = set(nova_log)
mira_skills = set(mira_log)
shared_skills = nova_skills & mira_skills
all_skills = nova_skills | mira_skills
nova_missing = required - nova_skills
mira_missing = required - mira_skills
nova_ready = required <= nova_skills
mira_ready = required <= mira_skills
team_signature = frozenset(all_skills)
display_skills = sorted(all_skills)
```

The logs preserve order and duplicates as evidence. Sets answer membership questions. The frozen set is a stable value that could become a mapping key, and the sorted list exists only to guarantee display order.

10. Explain the group model

1. Why does `set(log)` lose information even when it correctly deduplicates?
2. When should missing removal raise `KeyError`, and when is a no-op better?
3. Why is `a - b` different from `b - a`?
4. What fact does `required <= available` state?
5. Why can a `frozenset` be a dictionary key while an ordinary set cannot?

Key points

Key points

- Sets retain unique hashable members and do not promise numeric positions or a stable presentation order.
- `.add()` adds one member; `.update()` consumes members from iterables.
- `.remove()` requires a member, `.discard()` accepts absence, and `.pop()` removes an arbitrary member.
- Union, intersection, difference, and symmetric difference answer distinct group questions without mutating their operands.
- Subset, superset, and disjoint comparisons state readiness and overlap rules.
- `frozenset` provides immutable set membership and can be hashable itself.
- Keep an ordered source list when duplicates or arrival order remain meaningful. :::

References

- Python tutorial: sets
- Python standard types: set and frozenset
- Python built-in functions: `set` and `frozenset`
- Python glossary: hashable

Visiting Items in a Collection

Online lesson and runnable notebook

Trace what simple for loops receive from strings, lists, tuples, dictionaries, sets, ranges, enumerate, and zip while preserving order and alignment.

- **Level:** Python Foundations · Unit 3
- **Estimated time:** 3–4.5 hours
- **You will learn:** Trace each value supplied during simple collection traversal and use `range()`, `enumerate()`, and `zip()` without losing position or alignment.
- **Practice in:** Google Colab, JupyterLab, or a local editor

1. Visit every route stop without manual repetition

A route manifest contains stops in travel order:

```python
route = ["dock", "garden", "tower", "vault"]

for stop in route:
    print(f"Scanning {stop}")
```

The `for` statement asks `route` to supply items one at a time. Before each indented execution, Python binds the next item to `stop`. The list itself remains unchanged.

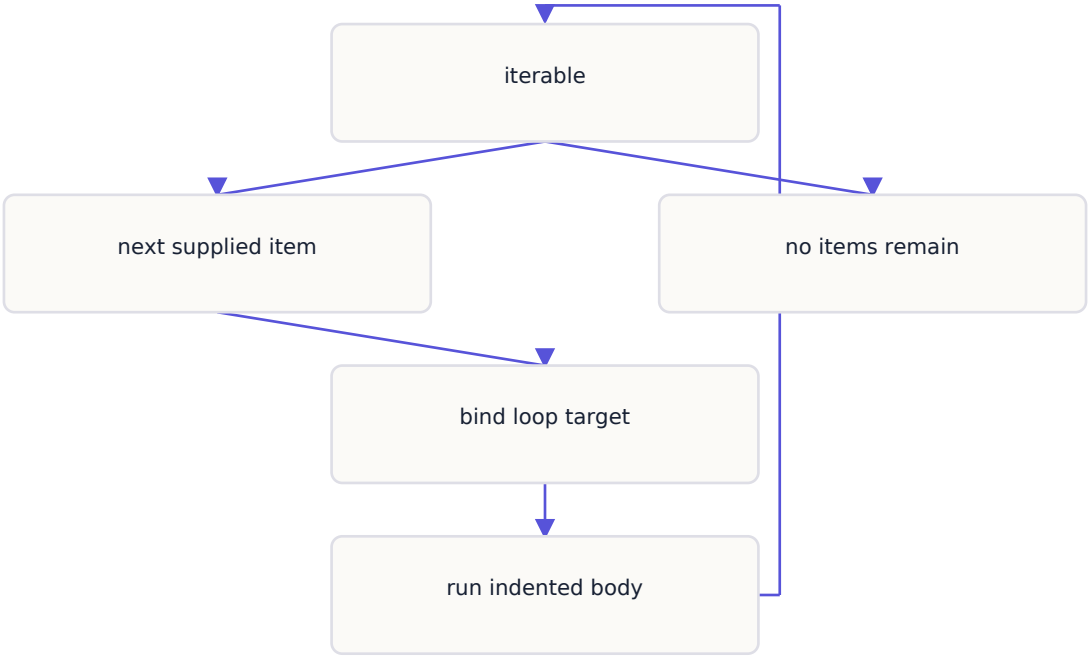
You could write four separate `print()` calls, but the repeated structure would break as soon as the route changed. Traversal connects one stated action to every currently supplied item.

An **iterable** is a value that can supply a sequence of items for this kind of visit. This lesson focuses on what each built-in collection supplies. Unit 5 later opens the iterator protocol and explains generators.

The practical questions are:

- What value does the loop target receive from each collection family?
- When do positions matter enough to use `range()` or `enumerate()`?
- How does `zip()` preserve alignment, and how can unequal lengths hide data?
- Why can structural mutation make an active traversal invalid or misleading?

A `for` loop repeatedly asks an iterable for a next item, binds the target name, and executes the body until no items remain.



2. Sequences supply values in sequence order

Lists, tuples, and strings preserve a deliberate order:

```python
items = ["map", "rope"]
coordinate = (12, 7)
code = "NOVA"

for item in items:
    print(item)

for number in coordinate:
    print(number)

for character in code:
    print(character)
```

The three loop targets receive strings, integers, and one-character strings, respectively. The target name is ordinary: after traversal it remains bound to the last supplied value.

```python
for character in "NOVA":
    print(character)

print(f"Last character: {character}")
```

That behavior is sometimes useful for tracing but usually should not be the way later code obtains the final item. `code[-1]` states that position directly.

An empty iterable runs the body zero times:

```python
empty_route = []
```

PYTHON (CONTINUED)

```
for stop in empty_route:
    print("This line does not run")

print("Traversal finished")
```

If `stop` had not existed before, using it after this empty loop would raise `NameError`. Do not rely on a loop target as the only source of a result that must exist for empty input.

Rebinding the target does not replace a sequence item

PYTHON

```
names = ["nova", "mira"]

for name in names:
    name = name.title()

print(names)
```

The list still contains lowercase strings. Assignment changes which string the local name `name` references; it does not assign through a list position.

3. Dictionaries can supply keys, values, or pairs

Direct traversal of a dictionary supplies its keys in insertion order:

PYTHON

```
energy_by_pilot = {"Nova": 80, "Mira": 65, "Sol": 90}

for pilot in energy_by_pilot:
    print(pilot)
```

Use the chosen key to retrieve a value:

PYTHON

```
for pilot in energy_by_pilot:
    energy = energy_by_pilot[pilot]
    print(f"{pilot}: {energy}")
```

Or ask a view to supply exactly what the body needs:

PYTHON

```
for energy in energy_by_pilot.values():
    print(energy)

for pilot, energy in energy_by_pilot.items():
    print(f"{pilot}: {energy}")
```

Each `.items()` value is a two-item tuple. Tuple unpacking assigns its key to `pilot` and value to `energy` before running the body.

PYTHON

```
pairs = list(energy_by_pilot.items())
print(pairs)
```

This makes the supplied shape visible.

4. Sets supply members without a positional promise

```python
skills = {"mapping", "repair", "translation"}

for skill in skills:
    print(skill)
```

Every current member is supplied once. The display order is not a stable program contract, so this would be fragile:

```python
observed_order = []

for skill in skills:
    observed_order.append(skill)

print(observed_order)
```

The traversal is valid, but comparing `observed_order` with one hard-coded order is not. Sort for presentation or preserve an ordered source list when sequence meaning matters.

Checkpoint: what each collection supplies

Collection traversal checkpoint

Identify the value bound to the target on each iteration.

1. What values does `for stop in ['dock', 'vault']` bind to `stop`?

- **A.** First `'dock'`, then `'vault'`
- **B.** First index 0, then index 1
- **C.** The complete list twice
- **D.** Only the final string

2. Which expression supplies key-value tuples from dictionary `d`?

- **A.** `d.values()`
- **B.** `d.items()`
- **C.** `d.keys()[0]`
- **D.** `d.index()`

3. How many times does the body of `for item in []:` execute?

- **A.** Once with `None`
- **B.** Once with an empty list
- **C.** Zero times
- **D.** Forever

4. Which fact is safe to rely on when traversing a set?

- **A.** The first inserted member is always first.
- **B.** Members are supplied alphabetically.

- **C.** Numeric indexes choose members.
- **D.** Every current member is supplied, but positional display order is not promised.

Write your choices before checking the answer key at the end of this book.

5. `range()` supplies an arithmetic progression

`range(stop)` starts at zero and excludes the stop:

```python
for index in range(4):
    print(index)
```

The values are `0`, `1`, `2`, and `3`. This aligns naturally with four list indexes, although direct item traversal is clearer when the body needs only each item.

`range(start, stop)` chooses the beginning:

```python
print(list(range(2, 6)))
```

The small inspection list is `[2, 3, 4, 5]`.

`range(start, stop, step)` chooses the distance and direction:

```python
evens = range(2, 11, 2)
countdown = range(5, 0, -1)

print(list(evens))
print(list(countdown))
```

The stop remains excluded. A negative step needs a start greater than the stop. Bounds pointing in the wrong direction produce an empty range:

```python
assert list(range(2, 8, -1)) == []
assert list(range(8, 2, 1)) == []
```

A zero step cannot make progress:

```python
invalid = range(1, 5, 0)
```

Python raises `ValueError: range() arg 3 must not be zero.`

A range is not first built as a large list

```python
positions = range(1_000_000)

print(type(positions).__name__)
print(len(positions))
print(positions[999_999])
```

The range stores its boundary rules compactly and calculates members as needed. Use `list(range(...))` to inspect only small progressions; converting a million positions defeats that advantage.

6. `enumerate()` adds positions to values

Manual bookkeeping creates two state changes to keep synchronized:

```python
route = ["dock", "garden", "vault"]
index = 0

for stop in route:
    print(index, stop)
    index = index + 1
```

`enumerate()` supplies `(position, value)` tuples directly:

```python
for index, stop in enumerate(route):
    print(index, stop)
```

Choose a custom start for human-facing numbering:

```python
for stop_number, stop in enumerate(route, start=1):
    print(f"Stop {stop_number}: {stop}")
```

The list indexes have not changed; `start=1` changes only the numbers supplied by `enumerate()`. Do not use `stop_number` to index the list unless you account for that offset.

Inspect the shape:

```python
numbered = list(enumerate(["dock", "vault"], start=1))
assert numbered == [(1, "dock"), (2, "vault")]
```

Checkpoint: range and enumerate

Range and enumerate checkpoint

Trace excluded stops, step direction, and supplied positions.

1. What values does `range(3)` supply?

- **A.** `0, 1, 2`
- **B.** `1, 2, 3`
- **C.** `0, 1, 2, 3`
- **D.** Only `3`

2. Which expression supplies `5, 4, 3, 2, 1`?

- **A.** `range(1, 5)`
- **B.** `range(5, 0, -1)`

- **C.** `range(5, 1)`
- **D.** `range(0, 5, -1)`

3. What does `enumerate(route)` supply on each iteration?

- **A.** Only a stop string
- **B.** Only a list index
- **C.** A tuple containing an index and the corresponding value
- **D.** A dictionary mapping every index at once

4. What does `start=1` change in `enumerate(route, start=1)`?

- **A.** It removes route index 0.
- **B.** It mutates the route.
- **C.** It starts traversal at the second item.
- **D.** It changes the counter values, not the underlying list positions.

Write your choices before checking the answer key at the end of this book.

7. `zip()` aligns values by traversal position

Two collections may describe the same stops:

```python
stops = ["dock", "garden", "vault"]
distances = [0, 4, 9]

for stop, distance in zip(stops, distances):
    print(f"{stop}: {distance} km")
```

`zip()` asks each input for one next item and supplies the grouped tuple. The first stop aligns with the first distance, then the second with the second.

```python
pairs = list(zip(stops, distances))
assert pairs == [("dock", 0), ("garden", 4), ("vault", 9)]
```

More than two iterables can be aligned:

```python
statuses = ["clear", "caution", "locked"]
rows = list(zip(stops, distances, statuses, strict=True))

assert rows[1] == ("garden", 4, "caution")
```

Default zip stops at the shortest input

```python
stops = ["dock", "garden", "vault"]
distances = [0, 4]

pairs = list(zip(stops, distances))
print(pairs)
```

The unpaired `"vault"` disappears from the result without an error. That can be useful when shortest-input behavior is intended, but dangerous when unequal lengths mean corrupted alignment.

Use strictness for equal-length data contracts:

```python
pairs = list(zip(stops, distances, strict=True))
```

Python raises `ValueError` when one iterable ends before the others.

Prefer records when fields belong together

Parallel lists are easy to misalign:

```python
stops = ["dock", "garden", "vault"]
distances = [0, 4, 9]
statuses = ["clear", "caution", "locked"]
```

A list of complete records keeps each relationship local:

```python
route_records = [
    {"stop": "dock", "distance": 0, "status": "clear"},
    {"stop": "garden", "distance": 4, "status": "caution"},
    {"stop": "vault", "distance": 9, "status": "locked"},
]
```

Use `zip(strict=True)` when parallel inputs arrive separately and equal length is a contract. Prefer records when the program owns the model and the values always travel together.

8. Do not invalidate an active traversal

Removing list items while visiting the same list can skip values because later items shift left:

```python
stops = ["dock", "closed", "closed", "vault"]

for stop in stops:
    if stop == "closed":
        stops.remove(stop)

print(stops)
```

One `"closed"` may remain. After the first removal, the second shifts into the position the loop has already passed.

When intentional removal is small and already decided, traverse an outer copy:

```python
stops = ["dock", "closed", "closed", "vault"]

for stop in stops.copy():
    if stop == "closed":
        stops.remove(stop)

assert stops == ["dock", "vault"]
```

Unit 4 develops clearer filtering patterns that build a new list rather than repeatedly removing.

Changing a dictionary's size during traversal raises a direct error:

```python
energy = {"Nova": 80, "Mira": 0}

for pilot in energy:
    if energy[pilot] == 0:
        del energy[pilot]
```

Python raises `RuntimeError: dictionary changed size during iteration`. Traverse a snapshot such as `list(energy)` for a deliberate small removal, or plan a separate result. Updating an existing value without changing dictionary size is permitted, but mixing traversal and mutation can still obscure the state transition.

Checkpoint: zip, alignment, and mutation

Alignment and mutation checkpoint

Preserve related values and keep traversal state valid.

1. What is the first tuple from `zip(['dock', 'vault'], [0, 9])`?

- **A.** `('dock', 0)`
- **B.** `('dock', 'vault')`
- **C.** `(0, 9)`
- **D.** `['dock', 0, 'vault', 9]`

2. How can code make unequal zip input lengths fail instead of silently truncating?

- **A.** Use `reverse=True`.
- **B.** Pass `strict=True` to `zip()`.
- **C.** Convert both inputs to sets.
- **D.** Start enumerate at one.

3. Why can one list of record dictionaries be safer than three parallel lists?

- **A.** Dictionaries always sort records.
- **B.** Records remove every duplicate.
- **C.** Each record keeps related fields together instead of relying on synchronized positions.
- **D.** Lists cannot contain strings.

4. Why can removing from a list while traversing it skip an adjacent matching item?

- **A.** remove() changes strings into sets.
- **B.** The loop runs backward.
- **C.** Python ignores duplicate values.
- **D.** Remaining items shift into positions the traversal has already advanced beyond.

Write your choices before checking the answer key at the end of this book.

9. Assemble the expedition route manifest

Build predictable display lines from related collections. Use the named helpers instead of manual index bookkeeping.

PYTHON

```
stops = ["dock", "garden", "tower", "vault"]
distances = [0, 4, 7, 11]
status_by_stop = {
    "dock": "clear",
    "garden": "caution",
    "tower": "clear",
    "vault": "locked",
}

numbered_stops = None
aligned_route = None
manifest_lines = []
status_lines = []
inspection_positions = None
```

Complete these stages:

1. make `numbered_stops` a list of `(1, stop)` tuples using `enumerate()`;
2. make `aligned_route` a list of stop-distance tuples with strict zip;
3. traverse the aligned tuples with human-facing numbers and append exactly the required manifest lines;
4. traverse dictionary items and append status lines in insertion order; and
5. create odd zero-based inspection positions below the route length with one `range()`.

PYTHON

```
assert numbered_stops == [
    (1, "dock"),
    (2, "garden"),
    (3, "tower"),
    (4, "vault"),
]
assert aligned_route == [
    ("dock", 0),
    ("garden", 4),
    ("tower", 7),
    ("vault", 11),
]
assert manifest_lines == [
    "Stop 1: dock (0 km)",
    "Stop 2: garden (4 km)",
    "Stop 3: tower (7 km)",
    "Stop 4: vault (11 km)",
]
assert status_lines == [
    "dock=clear",
    "garden=caution",
    "tower=clear",
    "vault=locked",
]
assert inspection_positions == [1, 3]
assert stops == ["dock", "garden", "tower", "vault"]
assert distances == [0, 4, 7, 11]
```

Boundary variation: remove the last distance and verify that strict zip fails. Restore the data, then add a complete fifth stop and distance and update the status mapping. A clean rerun should include it without another hard-coded append.

Hint: materialize only the values the assertions ask to retain

Use `list(enumerate(stops, start=1))` and `list(zip(stops, distances, strict=True))`. A second `enumerate(..., start=1)` can number the already aligned `(stop, distance)` tuples. Dictionary `.items()` supplies status pairs. `range(1, len(stops), 2)` supplies odd indexes.

Show one complete solution after attempting the lab

PYTHON

```
numbered_stops = list(enumerate(stops, start=1))
aligned_route = list(zip(stops, distances, strict=True))

manifest_lines = []
for stop_number, route_pair in enumerate(aligned_route, start=1):
    stop, distance = route_pair
    manifest_lines.append(f"Stop {stop_number}: {stop} ({distance} km)")

status_lines = []
for stop, status in status_by_stop.items():
    status_lines.append(f"{stop}={status}")

inspection_positions = list(range(1, len(stops), 2))
```

The first two results are retained because the contract asks for them. The output lists are built separately, so none of the source collections change.

10. Keep the boundary with Unit 4 clear

This lesson establishes what an iterable supplies and how values align. Unit 4 will build the fuller control-flow toolkit:

- choosing different actions with `if`, `elif`, and `else`;
- accumulating totals and filtered results;
- `break`, `continue`, and loop `else`;
- nested traversal and grid algorithms;
- condition-controlled `while` loops; and
- comprehensions.

You do not need those techniques to explain the collection contracts on this page.

Key points

Key points

- A simple `for` loop binds each item supplied by an iterable and runs its body.
- Sequences supply values in order; dictionaries directly supply keys; sets supply every current member without a positional-order promise.
- `range()` represents an excluded-stop arithmetic progression without first building a complete list.
- `enumerate()` supplies `(counter, value)` tuples and can start its counter at a custom number.

- `zip()` aligns inputs by traversal position; default zip truncates to the shortest input, while `strict=True` enforces equal exhaustion.
- Keep related fields in records when parallel positions are fragile.
- Avoid changing collection size while traversing that same structure. :::

References

- Python tutorial: `for` statements and `range`
- Python built-in function: `enumerate`
- Python built-in function: `zip`
- Python standard types: ranges
- Python glossary: iterable

Sorting Collections Without Losing Meaning

Online lesson and runnable notebook

Sort values and complete records by explicit policies while distinguishing mutation, new results, reversal, key functions, type boundaries, stability, and source preservation.

- **Level:** Python Foundations · Unit 3
- **Estimated time:** 3–4.5 hours
- **You will learn:** Sort scalar values and related record fields by an explicit, stable policy without accidentally replacing a list with `None` or destroying source order.
- **Practice in:** Google Colab, JupyterLab, or a local editor

1. Rank complete records, not disconnected fields

A tournament result connects a pilot, score, and completion time:

```python
results = [
    {"pilot": "Nova", "score": 91, "seconds": 74},
    {"pilot": "Mira", "score": 96, "seconds": 83},
    {"pilot": "Sol", "score": 91, "seconds": 68},
]
```

Sorting a separate score list would lose which pilot earned each score. The unit of movement must be the **complete record**. The sort policy then chooses which field determines order.

This lesson answers:

- Does the operation mutate the list or create a new list?
- Is reversal the same as sorting in descending order?
- How does a key function turn a domain rule into a comparable value?
- What happens when values are not mutually comparable?
- How does stable sorting preserve earlier order among equal keys?

2. `sorted()` creates a new list

The built-in `sorted()` accepts any iterable and returns a list:

```python
scores = [91, 74, 96, 83]
ranked_scores = sorted(scores)

print(scores)
print(ranked_scores)
```

`scores` remains in source order. `ranked_scores` is `[74, 83, 91, 96]`.

The input does not need to be a list:

```python
tuple_result = sorted((3, 1, 2))
set_result = sorted({"vault", "dock", "garden"})
dict_result = sorted({"Nova": 91, "Mira": 96})

print(type(tuple_result).__name__)
print(set_result)
print(dict_result)
```

All three results are lists. Direct dictionary traversal supplies keys, so `sorted(dictionary)` sorts keys.

3. `.sort()` changes one list and returns `None`

```python
scores = [91, 74, 96, 83]
result = scores.sort()

print(scores)
print(result)
```

The existing list changes and `result` is `None`. This makes a common bug clear:

```python
scores = [91, 74, 96]
scores = scores.sort()

print(scores)
```

The useful list name now refers to `None`. The sort already happened in place before assignment discarded the reference.

Choose from the state requirement:

- use `sorted(source)` when source order remains evidence or both versions matter;
- use `working.sort()` when the working list itself should adopt the new order.

Copy before an intentional in-place sort

```python
source_scores = [91, 74, 96, 83]
working_scores = source_scores.copy()
working_scores.sort()

assert source_scores == [91, 74, 96, 83]
assert working_scores == [74, 83, 91, 96]
```

This separates preservation from mutation visibly.

4. Descending sort and reversal answer different questions

Use `reverse=True` to sort in descending order:

```python
scores = [91, 74, 96, 83]
highest_first = sorted(scores, reverse=True)

assert highest_first == [96, 91, 83, 74]
```

`.reverse()` merely flips the current list positions:

```python
arrival_scores = [91, 74, 96, 83]
result = arrival_scores.reverse()

assert arrival_scores == [83, 96, 74, 91]
assert result is None
```

The result is not score-ranked; it is reverse arrival order.

`reversed()` supplies values in reverse traversal order without changing the source:

```python
route = ["dock", "garden", "vault"]
backtrack = list(reversed(route))

assert backtrack == ["vault", "garden", "dock"]
assert route == ["dock", "garden", "vault"]
```

`reversed(route)` itself is a reverse iterator, not a list. Convert it only when the program needs to retain all reversed values as a list.

Checkpoint: mutation and return values

Sorting state checkpoint

Track the original collection, returned value, and ordering rule.

1. What does `ranked = sorted(scores)` guarantee?

- **A.** `ranked` is a new sorted list and `scores` is not mutated.
- **B.** `ranked` is always a tuple.
- **C.** `scores` becomes `None`.
- **D.** Only sets can be sorted.

2. What does `result` reference after `result = scores.sort()`?

- **A.** A sorted copy
- **B.** `None`
- **C.** The original unsorted list
- **D.** The largest score

3. What does `items.reverse()` do?

- **A.** It sorts descending by value.
- **B.** It creates a sorted tuple.
- **C.** It reverses the current positions in place.
- **D.** It returns a reverse-ordered copy.

4. What type does `sorted({'b', 'a'})` return?

- **A.** `set`
- **B.** `frozenset`

- **C.** `tuple`
- **D.** `list`

Write your choices before checking the answer key at the end of this book.

5. Strings follow Unicode ordering unless you choose another policy

Default string comparison is case-sensitive and based on Unicode code points:

```python
pilots = ["nova", "Mira", "sol", "Ada"]
default_order = sorted(pilots)

print(default_order)
```

Uppercase spellings can appear before lowercase ones. That behavior is deterministic but may not match a human-facing name policy.

The `key` argument asks Python to calculate a comparison key for each value:

```python
name_order = sorted(pilots, key=str.casefold)
print(name_order)
```

`str.casefold` is passed without parentheses. Python calls it once per string for comparison and still returns the original spellings in the result.

```python
assert name_order == ["Ada", "Mira", "nova", "sol"]
```

This is a preview of a function used as a value. Unit 5 explains callables and callbacks fully. For now, read `key=str.casefold` as “compare the casefolded form of each original string.”

Other built-ins can express simple policies:

```python
words = ["star", "observatory", "map", "vault"]
by_length = sorted(words, key=len)

offsets = [-12, 4, -3, 9]
by_distance_from_zero = sorted(offsets, key=abs)

print(by_length)
print(by_distance_from_zero)
```

6. Mixed incomparable types should fail visibly

Python does not invent an ordering between unrelated types:

```python
mixed = [3, "12", 7]
print(sorted(mixed))
```

This raises `TypeError` because integer and string ordering is undefined. Do not silently sort everything by `str` unless lexicographic display text is genuinely the required policy:

```python
display_order = sorted(mixed, key=str)
print(display_order)
```

The key strings are `"3"`, `"12"`, and `"7"`, so this is not numeric order. A better data boundary often converts all inputs to one intentional type before storing them.

Missing values need an explicit policy too:

```python
scores = [91, None, 74]
print(sorted(scores))
```

The error asks you to decide whether `None` belongs first, last, or outside the ranked data. Sorting cannot infer domain meaning.

7. Sort records by fields with `itemgetter`

`operator.itemgetter()` creates a key callable that retrieves dictionary fields:

```python
from operator import itemgetter

results = [
    {"pilot": "Nova", "score": 91, "seconds": 74},
    {"pilot": "Mira", "score": 96, "seconds": 83},
    {"pilot": "Sol", "score": 91, "seconds": 68},
]

by_score = sorted(results, key=itemgetter("score"), reverse=True)

for record in by_score:
    print(record["pilot"], record["score"])
```

Complete dictionaries move together; no pilot becomes disconnected from a score or time.

`itemgetter()` can retrieve several fields as a tuple key:

```python
alphabetical_score_groups = sorted(
    results,
    key=itemgetter("score", "pilot"),
)
```

Python compares the score first and uses pilot only when scores are equal. The complete tuple key is ascending unless `reverse=True` reverses all of it.

An optional lambda spelling is common in Python code:

```python
by_time = sorted(results, key=lambda record: record["seconds"])
```

Read this as a tiny unnamed function that receives one record and returns its seconds. You do not need to write lambdas in this unit; `itemgetter()` and built-in functions cover the lessons. Unit 5 teaches function values and lambdas in context.

Checkpoint: key selection and type boundaries

Sort-key checkpoint

Choose a key that represents the stated domain order.

1. Which call sorts string names without treating case differences as the primary order?

- **A.** `sorted(names, key=str.casefold)`
- **B.** `sorted(names).casefold()`
- **C.** `names.sort(key=len)`
- **D.** `reversed(names)`

2. Which key sorts dictionary records by their `'score'` field?

- **A.** `key='score'`
- **B.** `key=itemgetter('score')`
- **C.** `key=records['score']`
- **D.** `key=dict.sort`

3. Why does `sorted([3, '12', 7])` raise `TypeError`?

- **A.** Lists cannot contain three values.
- **B.** Strings cannot be stored in lists.
- **C.** Python has no default ordering comparison between integers and strings.
- **D.** The list must be a tuple first.

4. Why can `sorted(numbers, key=str)` produce the wrong numeric policy?

- **A.** str() removes all digits.
- **B.** It mutates each number into a list.
- **C.** It always sorts descending.
- **D.** It compares text lexicographically, so `'12'` can appear before `'3'`.

Write your choices before checking the answer key at the end of this book.

8. Stable sorting preserves earlier decisions among ties

Python’s sort is **stable**: records with equal keys retain their relative order from the input.

```python
from operator import itemgetter

results = [
    {"pilot": "Nova", "score": 91, "arrival": 1},
    {"pilot": "Mira", "score": 96, "arrival": 2},
    {"pilot": "Sol", "score": 91, "arrival": 3},
    {"pilot": "Ivo", "score": 91, "arrival": 4},
]

ranked = sorted(results, key=itemgetter("score"), reverse=True)

assert [record["pilot"] for record in ranked] == [
```

PYTHON (CONTINUED)

```
    "Mira",
    "Nova",
    "Sol",
    "Ivo",
]
```

Nova, Sol, and Ivo all have score 91, so their arrival order remains intact. The list comprehension in this assertion is supplied only to inspect names; Unit 4 teaches comprehensions. An equivalent beginner-readable inspection is:

PYTHON

```
ranked_names = []

for record in ranked:
    ranked_names.append(record["pilot"])

assert ranked_names == ["Mira", "Nova", "Sol", "Ivo"]
```

Use stable passes for mixed directions

Suppose higher scores rank first, but equal scores use fewer seconds first. One tuple key with `reverse=True` would reverse both score and seconds, making slower times win ties. Stable passes express the mixed directions:

PYTHON

```
results = [
    {"pilot": "Nova", "score": 91, "seconds": 74},
    {"pilot": "Mira", "score": 96, "seconds": 83},
    {"pilot": "Sol", "score": 91, "seconds": 68},
    {"pilot": "Ivo", "score": 91, "seconds": 74},
]

ranked = sorted(results, key=itemgetter("seconds"))
ranked = sorted(ranked, key=itemgetter("score"), reverse=True)

ranked_names = []
for record in ranked:
    ranked_names.append(record["pilot"])

assert ranked_names == ["Mira", "Sol", "Nova", "Ivo"]
```

Apply the less important rule first (`seconds` ascending), then the most important rule (`score` descending). Stability preserves time order inside equal-score groups, and Nova remains ahead of Ivo because both key fields tie and Nova appeared first.

Sort dictionary items, not just keys

PYTHON

```
score_by_pilot = {"Nova": 91, "Mira": 96, "Sol": 91}
score_pairs = sorted(score_by_pilot.items(), key=itemgetter(1), reverse=True)

print(score_pairs)
```

Each item is `(key, value)`, so `itemgetter(1)` selects the score. Equal scores retain dictionary insertion order. The source dictionary remains unchanged.

Do not sort more often than the output requires

Sorting performs more work than one traversal. Sort once when the program needs an ordered view, retain the result while it remains valid, and avoid sorting the same unchanged collection inside another traversal. Unit 7 will give this comparison a formal vocabulary. :::

Checkpoint: stable record ordering

Stable record sorting checkpoint

Preserve record relationships and apply tie rules deliberately.

1. What does sort stability guarantee for records with equal keys?

- **A.** They retain their relative input order.
- **B.** They are removed as duplicates.
- **C.** They become dictionaries.
- **D.** They are ordered randomly.

2. Why should a leaderboard sort complete records rather than a separate score list?

- **A.** Scores cannot be integers.
- **B.** Each pilot, score, and time remains connected as one moving item.
- **C.** Dictionaries are automatically immutable.
- **D.** Separate score lists always raise errors.

3. For score descending and time ascending, which stable-pass order works?

- **A.** Score ascending, then time descending
- **B.** Score descending twice
- **C.** Time ascending first, then score descending
- **D.** Reverse the source without sorting

4. What should be sorted to retain both keys and values from a dictionary?

- **A.** Only `mapping.keys()`
- **B.** Only `mapping.values()`
- **C.** The dictionary length
- **D.** `mapping.items()` tuples

Write your choices before checking the answer key at the end of this book.

9. Produce the tournament leaderboard

Preserve the submitted result order and build two ranked views.

PYTHON

```
from operator import itemgetter

source_results = [
    {"pilot": "Nova", "score": 91, "seconds": 74},
    {"pilot": "Mira", "score": 96, "seconds": 83},
    {"pilot": "Sol", "score": 91, "seconds": 68},
```

PYTHON (CONTINUED)

```
    {"pilot": "Ivo", "score": 91, "seconds": 74},
]

alphabetical = None
ranked = None
leaderboard_lines = []
```

Requirements:

1. create an `alphabetical` list by pilot name without case-sensitive surprises;
2. create a `ranked` list where higher score wins, fewer seconds breaks a score tie, and original submission order breaks a complete tie;
3. append one display line per ranked record with a one-based place; and
4. leave the source list and every record unchanged.

PYTHON

```
assert [record["pilot"] for record in alphabetical] == [
    "Ivo",
    "Mira",
    "Nova",
    "Sol",
]
assert [record["pilot"] for record in ranked] == [
    "Mira",
    "Sol",
    "Nova",
    "Ivo",
]
assert leaderboard_lines == [
    "1. Mira — 96 points — 83s",
    "2. Sol — 91 points — 68s",
    "3. Nova — 91 points — 74s",
    "4. Ivo — 91 points — 74s",
]
assert source_results == [
    {"pilot": "Nova", "score": 91, "seconds": 74},
    {"pilot": "Mira", "score": 96, "seconds": 83},
    {"pilot": "Sol", "score": 91, "seconds": 68},
    {"pilot": "Ivo", "score": 91, "seconds": 74},
]
```

Boundary variation: append `{"pilot": "ada", "score": 96, "seconds": 83}` to the source. Predict its alphabetical position and how the complete leaderboard tie uses submission order. Rerun from the source cell rather than mutating an already ranked result.

Hint: separate the alphabetic and competition policies

`itemgetter("pilot")` is case-sensitive, so use an optional supplied lambda for the alphabetical view: `key=lambda record: record["pilot"].casefold()`. For mixed competition directions, sort by seconds ascending first and then by score descending. Enumerate the ranked records from one for display.

Show one complete solution after attempting the leaderboard

PYTHON

```
alphabetical = sorted(
    source_results,
    key=lambda record: record["pilot"].casefold(),
)
```

PYTHON (CONTINUED)

```
ranked = sorted(source_results, key=itemgetter("seconds"))
ranked = sorted(ranked, key=itemgetter("score"), reverse=True)

leaderboard_lines = []
for place, record in enumerate(ranked, start=1):
    line = (
        f"{place}. {record['pilot']} — "
        f"{record['score']} points — {record['seconds']}s"
    )
    leaderboard_lines.append(line)
```

Both `sorted()` calls create outer lists and move references to complete records; they do not mutate the source records. The stable second pass preserves the seconds rule and original order where the full ranking key ties.

10. Explain the ordering policy

1 Why is `items = items.sort()` a bug even though the sort itself occurs?

2 When is reverse traversal different from descending sorting?

3 What does a key callable return, and what values appear in the final list?

4 Why should mixed types or missing values trigger a policy decision?

5 How do stable passes express one descending and one ascending field?

Key points

Key points

- `sorted()` returns a new list; `.sort()` mutates a list and returns `None`.
- `reverse=True` reverses sort order; `.reverse()` and `reversed()` reverse current traversal order without calculating rank.
- A key callable produces the comparison value while complete original items move into the result.
- Normalize or reject mixed data deliberately; sorting cannot invent domain rules.
- Python sorting is stable, so equal-key items retain their earlier relative order.
- Sort complete records to preserve relationships, and use stable passes for mixed ascending and descending field requirements. :::

References

- Python sorting techniques
- Python built-in function: `sorted`
- Python tutorial: list method `sort`
- Python `operator.itemgetter`
- Python built-in function: `reversed`

Building and Reading Nested Data

Online lesson and runnable notebook

Choose and navigate nested collection shapes one level at a time, diagnose wrong-key and wrong-index failures, and build alternate lookup and grouping views without losing source relationships.

- **Level:** Python Foundations · Unit 3
- **Estimated time:** 3.5–5 hours
- **You will learn:** Describe, navigate, update, and refactor nested collection shapes while distinguishing wrong types, missing fields, missing positions, optional values, and shallow copies.
- **Practice in:** Google Colab, JupyterLab, or a local editor

1. Model one game world with several collection jobs

Real data rarely fits one flat collection. A game world may preserve region order, name each region’s fields, keep one fixed coordinate, group unique hazards, and store a nested weather record:

```python
world = [
    {
        "id": "garden",
        "coordinate": (4, 7),
        "hazards": {"fog"},
        "weather": {"condition": "mist", "temperature": 12},
    },
    {
        "id": "vault",
        "coordinate": (9, 2),
        "hazards": {"lock", "darkness"},
        "weather": {"condition": "dry", "temperature": 18},
    },
]
```

Read its type shape from the outside inward:

```text
list
└── dictionary region record
    ├── "id" -> string
    ├── "coordinate" -> tuple of two integers
    ├── "hazards" -> set of strings
    └── "weather" -> dictionary
        ├── "condition" -> string
        └── "temperature" -> integer
```

No single container is doing every job. The outer list preserves region display order. Each dictionary names fields. The tuple preserves a fixed coordinate, the set preserves unique hazards, and the nested dictionary names weather fields.

This lesson answers:

- How can you read a long path without guessing which type comes next?
- What do `TypeError`, `KeyError`, and `IndexError` reveal about the failed level?

- How should absent, stored-as-`None`, and empty values remain distinct?
- When should an ordered list of records gain a separate direct-lookup dictionary?

2. Navigate one container level at a time

The compact access below reaches the vault temperature:

```python
temperature = world[1]["weather"]["temperature"]
print(temperature)
```

Read every operation against its immediate left-hand value:

1. `world[1]` indexes the outer list and produces a region dictionary;
2. `...["weather"]` looks up a key and produces a weather dictionary; and
3. `...["temperature"]` looks up a key and produces integer `18`.

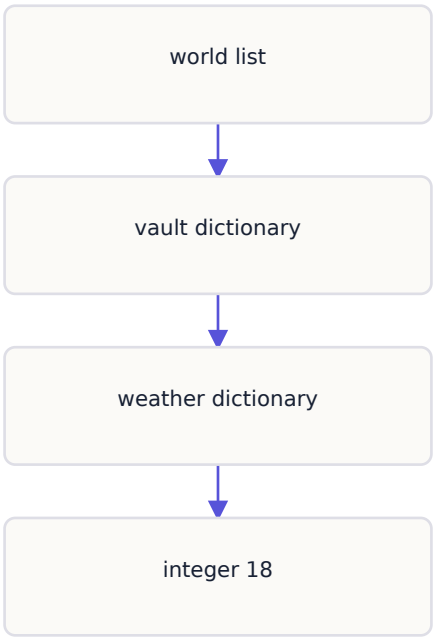
When the path is unfamiliar, store the intermediate levels:

```python
vault_record = world[1]
vault_weather = vault_record["weather"]
vault_temperature = vault_weather["temperature"]

print(type(vault_record).__name__)
print(type(vault_weather).__name__)
print(type(vault_temperature).__name__)
```

Intermediate names turn an opaque chain into inspectable evidence. They also give tracebacks a smaller region of code to identify.

Each access operation selects one level and produces the container or scalar needed by the next operation.



world list → vault dictionary: index 1

vault dictionary → weather dictionary: weather key

weather dictionary → integer 18: temperature key

Draw the path before writing it

For each requested result, state the levels:

Question	Access path	Types crossed
First region ID	`world[0]["id"]`	list → dictionary → string
Vault row	`world[1]["coordinate"][0]`	list → dictionary → tuple → integer
Garden hazards	`world[0]["hazards"]`	list → dictionary → set
Garden condition	`world[0]["weather"]["condition"]`	list → dictionary → dictionary → string

An index belongs to a sequence. A key belongs to a mapping. Membership belongs to a set-like group. Choosing the next operation depends on the value produced at the current level.

3. Wrong operations produce different evidence

Applying a string key to the outer list raises `TypeError`:

```
PYTHON
region = world["vault"]
```

The outer value is a list and expects an integer or slice. The error does not mean the key `"vault"` is misspelled; it means this **level has the wrong operation**.

A missing dictionary field raises `KeyError`:

```
PYTHON
terrain = world[0]["terrain"]
```

The region record is a dictionary, but it contains no `"terrain"` key.

An absent sequence position raises `IndexError`:

```
PYTHON
third_region = world[2]
```

The outer list has only positions 0 and 1.

A set rejects numeric indexing with `TypeError`:

```
PYTHON
first_hazard = world[1]["hazards"][0]
```

The path reaches the right field, but a set has no positional member zero.

Use the exception to identify the failed level

- `TypeError` often means the current value does not support the attempted kind of access.

- `KeyError` means mapping access reached a dictionary but the requested key is absent.
- `IndexError` means sequence access reached a sequence but the position is absent.

Print or inspect the value immediately to the left of the failing brackets before changing the full chain.
:::

Checkpoint: reading nested paths

Nested-path checkpoint

Follow one produced type at a time.

1. Which expression reads the temperature from the first record in `world`?

- **A.** `world[0]['weather']['temperature']`
- **B.** `world['weather'][0]['temperature']`
- **C.** `world[0][0]['temperature']`
- **D.** `world.weather.temperature`

2. What type does `world[1]['coordinate']` produce in the example?

- **A.** A dictionary
- **B.** A tuple
- **C.** A set
- **D.** An integer

3. Why does `world['vault']` fail when `world` is a list?

- **A.** The word vault is reserved.
- **B.** The list is empty.
- **C.** A list expects an integer index or slice, not a string key.
- **D.** Nested dictionaries cannot be stored in lists.

4. Which failure normally indicates that a dictionary record lacks the requested field?

- **A.** `IndexError`
- **B.** `NameError`
- **C.** `ZeroDivisionError`
- **D.** `KeyError`

Write your choices before checking the answer key at the end of this book.

4. Choose a nested shape from the dominant question

Several shapes can store the same facts, but they optimize different access questions.

A list of dictionaries preserves record order

```python
regions = [
    {"id": "garden", "temperature": 12},
    {"id": "vault", "temperature": 18},
]
```

Use this when ordered display or repeated full-record traversal matters. Finding one ID requires a search unless another index is built.

A dictionary of dictionaries gives direct ID lookup

```python
regions_by_id = {
    "garden": {"temperature": 12},
    "vault": {"temperature": 18},
}

print(regions_by_id["vault"]["temperature"])
```

Use this when stable IDs are unique and direct lookup dominates. The ID can live only as the outer key or be repeated inside each record when complete standalone records are valuable; choose one consistent contract.

A dictionary of lists groups values

```python
regions_by_climate = {
    "cool": ["garden", "tower"],
    "warm": ["vault"],
}
```

Use this when the central question is “Which region IDs belong to this group?” The list retains group order and can contain several members.

A list of lists models a position-based grid

```python
grid = [
    ["dock", "water", "water"],
    ["path", "garden", "wall"],
    ["path", "tower", "vault"],
]

print(grid[2][1])
```

The result is `"tower"`: choose row 2 from the outer list, then column 1 from that row. A rectangular grid is a good positional shape only when rows and columns have consistent meaning.

No shape wins universally. Preserve more than one view when the program has both ordered-display and repeated-direct-lookup requirements, and define which view is the source of truth.

5. Update the exact nested container you intend

Assignment through a complete path mutates the reached dictionary:

```python
world = [
    {
        "id": "garden",
        "weather": {"condition": "mist", "temperature": 12},
    },
    {
        "id": "vault",
        "weather": {"condition": "dry", "temperature": 18},
    },
]

world[0]["weather"]["temperature"] = 13
assert world[0]["weather"]["temperature"] == 13
```

The outer list still has two positions. The first region dictionary still has the same fields. The nested weather dictionary's temperature value changed.

Intermediate names can make the mutation target clearer:

```python
garden = world[0]
garden_weather = garden["weather"]
garden_weather["condition"] = "clear"

assert world[0]["weather"]["condition"] == "clear"
```

`garden_weather` is an alias for the nested dictionary; it is not a detached copy.

Outer copies still share nested values

```python
source_record = {
    "id": "garden",
    "weather": {"condition": "mist", "temperature": 12},
}
working_record = source_record.copy()

working_record["weather"]["temperature"] = 13

print(source_record["weather"]["temperature"])
```

The source also reports 13 because `.copy()` separated only the outer dictionary. For this one known shape, reconstruct the nested level deliberately:

```python
source_record = {
    "id": "garden",
    "weather": {"condition": "mist", "temperature": 12},
}
working_record = source_record.copy()
working_record["weather"] = source_record["weather"].copy()
working_record["weather"]["temperature"] = 13

assert source_record["weather"]["temperature"] == 12
assert working_record["weather"]["temperature"] == 13
```

This is not a universal deep-copy recipe. Unit 6 teaches shared reference graphs, `copy.deepcopy()`, and when rebuilding a value is preferable.

6. Keep absent, `None`, and empty values distinct

Consider three records:

```python
records = [
    {"id": "garden"},
    {"id": "vault", "guide": None},
    {"id": "tower", "guide": "Nova", "hazards": []},
]
```

- `garden` has no `"guide"` field;
- `vault` has a guide field whose value is deliberately missing; and
- `tower` has a guide name and an empty hazards list.

Membership and `.get()` serve different questions:

```python
garden, vault, tower = records

assert "guide" not in garden
assert "guide" in vault and vault["guide"] is None
assert tower.get("guide") == "Nova"
assert "hazards" in tower and tower["hazards"] == []
```

`.get("guide")` alone returns `None` for both the absent garden field and the stored-`None` vault field. Check membership when the schema state matters.

A default should match the expected type:

```python
garden_hazards = garden.get("hazards", [])
assert garden_hazards == []
```

Do not mutate a shared default object stored elsewhere. The fresh literal above is used only as a read fallback.

7. Consistent record shapes make traversal predictable

This collection promises that each record has an ID and coordinate:

```python
regions = [
    {"id": "garden", "coordinate": (4, 7)},
    {"id": "vault", "coordinate": (9, 2)},
]

for region in regions:
    region_id = region["id"]
    row, column = region["coordinate"]
    print(region_id, row, column)
```

A malformed record exposes the broken promise:

```python
regions = [
    {"id": "garden", "coordinate": (4, 7)},
    {"name": "vault", "coordinate": (9,)},
```

PYTHON (CONTINUED)

```
]

for region in regions:
    region_id = region["id"]
    row, column = region["coordinate"]
    print(region_id, row, column)
```

The second record first raises `KeyError` for `"id"`; after that key is repaired, its one-item coordinate raises an unpacking `ValueError`. These are separate schema failures, and both are more useful than guessing around them with unrelated defaults.

External validation belongs in Unit 9. Inside this lesson, write and assert the shape that your own in-memory examples promise.

Checkpoint: shape failures and optional values

Nested-shape boundary checkpoint

Preserve the difference between operation errors and meaningful missing states.

1. For a rectangular list-of-lists named `grid`, what does `grid[2][1]` do?

- **A.** Select row index 2, then column index 1
- **B.** Select a dictionary key named 2
- **C.** Select every row except 1
- **D.** Sort the grid by two fields

2. How do you prove key `'guide'` exists and deliberately stores `None`?

- **A.** `record.get('guide') is None` only
- **B.** `'guide' in record and record['guide'] is None`
- **C.** `record == None`
- **D.** `record.count('guide') == 0`

3. Why can an intermediate `weather = record['weather']` name mutate the original record's weather?

- **A.** String keys copy every value.
- **B.** All dictionaries are global.
- **C.** The name refers to the same nested dictionary object.
- **D.** Nested mutation changes only printed output.

4. What does `row, column = (9,)` reveal?

- **A.** Coordinates must be dictionaries.
- **B.** The row must be text.
- **C.** Tuple values cannot be assigned.
- **D.** The record's coordinate length does not match the promised two-position shape.

Write your choices before checking the answer key at the end of this book.

8. Avoid parallel lists when fields form records

This representation depends on matching positions forever:

```python
region_ids = ["garden", "vault", "tower"]
temperatures = [12, 18, 9]
conditions = ["mist", "dry", "wind"]
```

Removing `"vault"` from only one list silently assigns the wrong temperature and condition to later IDs. Strict zip detects length mismatch but cannot detect equal- length lists that are already semantically misaligned.

Keep fields together:

```python
regions = [
    {"id": "garden", "temperature": 12, "condition": "mist"},
    {"id": "vault", "temperature": 18, "condition": "dry"},
    {"id": "tower", "temperature": 9, "condition": "wind"},
]
```

Now one insertion, removal, or sort moves a complete record.

9. Add a lookup view without discarding source order

An ordered source and keyed index can coexist:

```python
regions = [
    {"id": "garden", "temperature": 12},
    {"id": "vault", "temperature": 18},
    {"id": "tower", "temperature": 9},
]

regions_by_id = {}

for region in regions:
    regions_by_id[region["id"]] = region

assert regions_by_id["vault"]["temperature"] == 18
assert regions[0]["id"] == "garden"
```

The list remains the ordered source. The dictionary is a secondary index for direct lookup. Its values intentionally reference the same record objects. If the program permits mutations, document which view owns changes so the structures do not drift.

Group IDs for another question:

```python
regions = [
    {"id": "garden", "climate": "cool"},
    {"id": "vault", "climate": "warm"},
    {"id": "tower", "climate": "cool"},
]
ids_by_climate = {}

for region in regions:
    climate = region["climate"]
    if climate not in ids_by_climate:
```

PYTHON (CONTINUED)

```
        ids_by_climate[climate] = []
    ids_by_climate[climate].append(region["id"])

assert ids_by_climate == {
    "cool": ["garden", "tower"],
    "warm": ["vault"],
}
```

One source can support several derived views, each answering a named question.

10. JSON-like does not mean JSON has been parsed

Lists, dictionaries, strings, numbers, Booleans, and `None` resemble the values commonly produced by parsing JSON:

PYTHON

```
json_like_value = {
    "regions": [
        {"id": "garden", "open": True},
        {"id": "vault", "open": False},
    ],
    "next_page": None,
}
```

This is an ordinary in-memory Python value. No file was opened and no external text was parsed. Unit 9 teaches JSON parsing, file encodings, schemas, and validation at an external-data boundary. The collection skills here prepare you to inspect the resulting shape safely.

Checkpoint: choosing and refactoring shapes

Nested collection choice checkpoint

Choose the top-level shape from the program’s dominant access question.

1. Which shape best preserves an ordered series of named region records?

- **A.** A list of dictionaries
- **B.** A set of lists
- **C.** One string containing commas
- **D.** A dictionary whose keys are mutable lists

2. Which shape directly answers `regions_by_id['vault']`?

- **A.** A list of temperatures
- **B.** A dictionary mapping IDs to complete records
- **C.** A set of region dictionaries
- **D.** A tuple of unrelated fields

3. What relationship can equal-length parallel lists still fail to preserve?

- **A.** Whether integers can be printed
- **B.** Whether each list has indexes
- **C.** Whether values at matching positions still describe the same record

- **D.** Whether strings are hashable

4. Why keep both an ordered source list and a dictionary lookup view?

- **A.** Every program must duplicate all data.
- **B.** Dictionaries cannot contain records.
- **C.** Lists cannot be traversed.
- **D.** They answer different questions: ordered display and direct access by stable ID.

Write your choices before checking the answer key at the end of this book.

11. Build the game-world archive

Preserve the supplied source tuple. Build ordered display, direct lookup, unique hazard, and climate-group views without mutating any record.

PYTHON

```
source_world = (
    {
        "id": "garden",
        "coordinate": (4, 7),
        "climate": "cool",
        "hazards": {"fog"},
        "guide": "Nova",
    },
    {
        "id": "vault",
        "coordinate": (9, 2),
        "climate": "warm",
        "hazards": {"lock", "darkness"},
        "guide": None,
    },
    {
        "id": "tower",
        "coordinate": (2, 8),
        "climate": "cool",
        "hazards": set(),
    },
)

region_ids = []
regions_by_id = {}
all_hazards = set()
ids_by_climate = {}
guide_states = {}
vault_row = None
vault_column = None
```

Requirements:

1. preserve source order in `region_ids`;
2. map each stable ID to its complete record;
3. combine every unique hazard;
4. group IDs by climate while preserving source order within each group;
5. classify each guide state as `"named"`, `"none"`, or `"absent"` using the supplied scaffold below; and
6. unpack the vault coordinate into named row and column values.

The classification uses simple conditional syntax supplied from the next unit:

```python
for region in source_world:
    region_id = region["id"]

    if "guide" not in region:
        guide_states[region_id] = "absent"
    elif region["guide"] is None:
        guide_states[region_id] = "none"
    else:
        guide_states[region_id] = "named"
```

Complete the other views around that scaffold, then run:

```python
assert region_ids == ["garden", "vault", "tower"]
assert list(regions_by_id) == ["garden", "vault", "tower"]
assert regions_by_id["garden"]["coordinate"] == (4, 7)
assert all_hazards == {"fog", "lock", "darkness"}
assert ids_by_climate == {
    "cool": ["garden", "tower"],
    "warm": ["vault"],
}
assert guide_states == {
    "garden": "named",
    "vault": "none",
    "tower": "absent",
}
assert (vault_row, vault_column) == (9, 2)
assert source_world == (
    {
        "id": "garden",
        "coordinate": (4, 7),
        "climate": "cool",
        "hazards": {"fog"},
        "guide": "Nova",
    },
    {
        "id": "vault",
        "coordinate": (9, 2),
        "climate": "warm",
        "hazards": {"lock", "darkness"},
        "guide": None,
    },
    {
        "id": "tower",
        "coordinate": (2, 8),
        "climate": "cool",
        "hazards": set(),
    },
)
```

Boundary variation: add a fourth well-shaped cool region with one repeated and one new hazard. Predict which output collections preserve the duplicate, discard it, or add another ordered ID before rerunning.

Hint: let every derived collection answer one named question

In one traversal, append the ID, assign the complete record by ID, update the hazard set from the record’s set, and create/append the climate group list. Use the provided guide-state scaffold in that same traversal or a second one. Retrieve the vault record from the finished lookup and unpack its coordinate.

Show one complete solution after attempting the archive

```python
region_ids = []
regions_by_id = {}
all_hazards = set()
ids_by_climate = {}
guide_states = {}

for region in source_world:
    region_id = region["id"]
    climate = region["climate"]

    region_ids.append(region_id)
    regions_by_id[region_id] = region
    all_hazards.update(region["hazards"])

    if climate not in ids_by_climate:
        ids_by_climate[climate] = []
    ids_by_climate[climate].append(region_id)

    if "guide" not in region:
        guide_states[region_id] = "absent"
    elif region["guide"] is None:
        guide_states[region_id] = "none"
    else:
        guide_states[region_id] = "named"

vault_row, vault_column = regions_by_id["vault"]["coordinate"]
```

Every derived collection has a distinct purpose. The lookup intentionally shares the immutable-by-convention source records, and the code never mutates them. The hazard view stores only unique strings, while the ordered ID and grouped-ID lists preserve the order relevant to their displays.

12. Explain the data shape

1. How do you determine whether the next brackets need an index or a key?
2. What different broken promises do `TypeError`, `KeyError`, and `IndexError` reveal along a nested path?
3. Why are absent, stored `None`, and empty collection values not interchangeable?
4. How can a shallow outer copy still expose source data to nested mutation?
5. When is a secondary dictionary lookup worth keeping beside an ordered list?

Key points

Key points

- Describe the outer type and every nested level before writing a long access path.
- Apply each index or key to the value immediately on its left; store intermediate values when the current type is unclear.
- A list of records, keyed record dictionary, grouped dictionary of lists, and position-based grid answer different access questions.
- Preserve absent, stored `None`, and empty values when they express different states.
- Consistent record fields and tuple lengths make traversal predictable; failures provide evidence about the malformed level.

- Parallel lists can drift; complete records preserve field relationships.
- A shallow copy separates an outer container but can retain nested aliases.
- Ordered source and direct-lookup views can coexist when ownership is explicit. :::

References

- Python tutorial: nested list comprehensions and nested structures
- Python tutorial: dictionaries
- Python standard types: sequence operations
- Python standard types: dictionary operations

Unit Challenge: Decode the Starship Signal Vault

Online lesson and runnable notebook

Preserve a damaged signal archive while building keyed, unique, grouped, ranked, and ordered views that reveal the vault code NOVA.

- **Level:** Python Foundations · Unit 3 challenge
- **Estimated time:** 60–120 minutes
- **Outcome assessed:** Choose, traverse, compare, sort, and combine collection shapes without losing source order, record relationships, unique membership, or fixed clue positions.
- **Evidence:** Twenty progressive checks, a decoded vault artifact, one boundary rerun, and one debugging record

1. Challenge outcome

The research starship *Wayfinder* has recovered five signals from a damaged archive. Four active transmissions contain the pieces of a vault code. One inactive transmission is interference, two signals have equal strength, several symbols repeat, and the records arrived in an order that must remain evidence.

Build several deliberate views of the archive and use the tuple clues to unlock the final word **NOVA**. The puzzle is solved only when the source remains intact, every progressive check passes unchanged, and the final artifact is produced from the derived collections.

Mission rule

Run one assertion group at a time. Repair the first failed promise before moving on. Do not edit expected values, sort the source list, replace sets with display lists, or hard-code the final vault word.
:::

The challenge uses simple `for` and `if` scaffolding because Unit 4 owns complete loop design. Your work is to choose the collection operations, preserve their relationships, and explain the state each derived view represents.

2. Understand the acceptance example

The final artifact must be exactly:

```text
=== WAYFINDER SIGNAL VAULT ===
Status: OPEN
Code: NOVA
Signals: 4 active / 5 total
Sectors: Lyra=2, Orion=2, Vela=1
Symbols: key, moon, star, sun
```

The code is not written directly in source order. Each active record contains a tuple `(code_position, letter)`. Sort complete clue tuples by position and then unpack their letters.

Collection contracts

1. `source_signals` remains the ordered source of truth.
2. `signals_by_id` maps every unique signal ID to its complete record.
3. `active_ids` preserves transmission order while excluding inactive records.
4. `active_symbols` contains unique symbols from active records only.
5. `required_symbols` remains a `frozenset` and determines `vault_ready`.
6. `sector_counts` counts all records, including the inactive one.
7. `clue_pairs` preserves active transmission order before sorting.
8. `ranked_signals` is a new list ordered by strength descending and ID ascending for equal strengths.
9. `sorted_clues` orders the tuple clues by code position.
10. `vault_word` is joined from derived letters, never typed as a solution value.

Constraints

- Keep every public variable name in the starter contract.
- Do not mutate any dictionary, set, tuple, or the outer source list.
- Do not use functions, comprehensions, exception handlers, classes, or nested loops.
- One straightforward traversal may build the lookup, active views, counts, and clue list together.
- Use tuple unpacking for every clue.
- Use set operations for symbol readiness.
- Use `operator.itemgetter()` and stable sorting for ranking.
- Sort only derived lists, never `source_signals` in place.

3. Start from the contract

Copy this complete starter into a fresh cell. `source_snapshot` is supplied so the checks can detect accidental nested mutation.

PYTHON

```
from operator import itemgetter

source_signals = [
    {
        "id": "SIG-N",
        "sector": "Lyra",
        "strength": 88,
        "status": "active",
        "clue": (0, "N"),
        "symbols": {"star", "moon"},
    },
    {
        "id": "SIG-X",
        "sector": "Orion",
        "strength": 99,
        "status": "inactive",
        "clue": (9, "X"),
        "symbols": {"comet"},
    },
```

PYTHON (CONTINUED)

```
    {
        "id": "SIG-O",
        "sector": "Lyra",
        "strength": 94,
        "status": "active",
        "clue": (1, "O"),
        "symbols": {"moon", "key"},
    },
    {
        "id": "SIG-A",
        "sector": "Vela",
        "strength": 82,
        "status": "active",
        "clue": (3, "A"),
        "symbols": {"sun", "key"},
    },
    {
        "id": "SIG-V",
        "sector": "Orion",
        "strength": 94,
        "status": "active",
        "clue": (2, "V"),
        "symbols": {"star", "sun"},
    },
]

source_snapshot = [
    {
        "id": "SIG-N",
        "sector": "Lyra",
        "strength": 88,
        "status": "active",
        "clue": (0, "N"),
        "symbols": {"star", "moon"},
    },
    {
        "id": "SIG-X",
        "sector": "Orion",
        "strength": 99,
        "status": "inactive",
        "clue": (9, "X"),
        "symbols": {"comet"},
    },
    {
        "id": "SIG-O",
        "sector": "Lyra",
        "strength": 94,
        "status": "active",
        "clue": (1, "O"),
        "symbols": {"moon", "key"},
    },
    {
        "id": "SIG-A",
        "sector": "Vela",
        "strength": 82,
        "status": "active",
        "clue": (3, "A"),
        "symbols": {"sun", "key"},
    },
    {
        "id": "SIG-V",
        "sector": "Orion",
        "strength": 94,
        "status": "active",
        "clue": (2, "V"),
        "symbols": {"star", "sun"},
    },
]
```

PYTHON (CONTINUED) (CONTINUED)

```
required_symbols = frozenset({"key", "moon", "star", "sun"})

# Stage A: archive views.
signals_by_id = {}
active_ids = []
active_symbols = set()
sector_counts = {}
clue_pairs = []

# Stage B: readiness, ranking, and decoding.
missing_symbols = None
vault_ready = None
ranked_signals = None
ranked_ids = []
sorted_clues = None
vault_letters = []
vault_word = None

# Stage C: final display values.
sector_text = None
symbol_text = None
artifact = None
```

4. Build in small stages

Stage A: create archive views in one traversal

Use this supplied structure. Replace each `None` with the relevant collection operation; do not add another nested loop.

PYTHON

```
for signal in source_signals:
    signal_id = signal["id"]
    sector = signal["sector"]

    # Map the stable ID to the complete signal record.
    signals_by_id[signal_id] = None

    # Count every record in its sector.
    sector_counts[sector] = None

    if signal["status"] == "active":
        # Preserve active transmission order.
        active_ids.append(None)

        # Combine the members of this record's symbol set.
        active_symbols.update(None)

        # Unpack the fixed clue tuple, then preserve it as one tuple.
        code_position, code_letter = signal["clue"]
        clue_pairs.append(None)
```

Run assertion group A. If a value has the wrong nesting level, inspect one signal record and the affected derived collection before changing another line.

Stage B: compare, rank, and decode

1 Calculate symbols missing from the required frozen set.

2 Calculate readiness with a subset relationship.

3 Rank **all** signals by ID ascending first, then strength descending. The second stable pass makes strength primary while retaining ID order among ties.

4 Traverse the ranked records to preserve their IDs in `ranked_ids`.

5 Sort the clue tuples. Because their first values are unique integer positions, ordinary tuple ordering gives the intended sequence.

6 Traverse `sorted_clues`, unpack each tuple, and append only its letter.

7 Join the letters into `vault_word`.

Run group B before formatting any display text.

Stage C: produce the vault artifact

The expected sector order is alphabetical. The expected symbol display is also alphabetical. Build:

```python
sector_text = (
    f"Lyra={sector_counts['Lyra']}, "
    f"Orion={sector_counts['Orion']}, "
    f"Vela={sector_counts['Vela']}"
)
symbol_text = ", ".join(sorted(active_symbols))
```

Use an adjacent multi-line f-string for `artifact`. Derive the status without an `if` expression by using the supplied Boolean tuple lookup:

```python
vault_status = ("SEALED", "OPEN")[vault_ready]
```

Do not type `NOVA` anywhere in the implementation.

5. Run progressive assertions

Group A: source, lookup, active order, counts, and clues

```python
assert source_signals == source_snapshot
assert len(signals_by_id) == 5
assert list(signals_by_id) == ["SIG-N", "SIG-X", "SIG-O", "SIG-A", "SIG-V"]
assert signals_by_id["SIG-O"]["strength"] == 94
assert active_ids == ["SIG-N", "SIG-O", "SIG-A", "SIG-V"]
assert "SIG-X" not in active_ids
assert sector_counts == {"Lyra": 2, "Orion": 2, "Vela": 1}
assert clue_pairs == [(0, "N"), (1, "O"), (3, "A"), (2, "V")]
```

Group B: unique membership, ranking, and decoded values

```python
assert active_symbols == {"key", "moon", "star", "sun"}
assert type(required_symbols) is frozenset
assert missing_symbols == set()
assert vault_ready is True
assert ranked_ids == ["SIG-X", "SIG-O", "SIG-V", "SIG-N", "SIG-A"]
assert source_signals == source_snapshot
assert sorted_clues == [(0, "N"), (1, "O"), (2, "V"), (3, "A")]
assert vault_letters == ["N", "O", "V", "A"]
assert vault_word == "".join(vault_letters)
```

Group C: final artifact

```python
expected_artifact = (
    "=== WAYFINDER SIGNAL VAULT ===\n"
    "Status: OPEN\n"
    "Code: NOVA\n"
    "Signals: 4 active / 5 total\n"
    "Sectors: Lyra=2, Orion=2, Vela=1\n"
    "Symbols: key, moon, star, sun"
)

assert sector_text == "Lyra=2, Orion=2, Vela=1"
assert symbol_text == "key, moon, star, sun"
assert artifact == expected_artifact
print(artifact)
```

A plausible result can still violate the contract

Typing `vault_word = "NOVA"`, sorting `source_signals` in place, or adding inactive `"comet"` to the active symbol set may make one visible line look plausible while earlier evidence is wrong. The unchanged progressive checks are part of the artifact. :::

6. Use the hint ladder only when needed

Hint 1

Match each question to one collection behavior:

- ID lookup: dictionary assignment with `signal_id` as key;
- active transmission order: list append inside the active branch;
- unique active symbols: set update from each active symbol set;
- sector frequency: `.get(sector, 0) + 1`; and
- fixed clue meaning: unpack `(code_position, code_letter)` and append that pair.

Run group A before attempting ranking.

Hint 2

The central Stage B operations have these shapes:

```python
missing_symbols = required_symbols - active_symbols
vault_ready = required_symbols <= active_symbols

ranked_signals = sorted(source_signals, key=itemgetter("id"))
ranked_signals = sorted(
    ranked_signals,
    key=itemgetter("strength"),
    reverse=True,
)

sorted_clues = sorted(clue_pairs)
```

Use separate straightforward traversals to append ranked IDs and unpacked clue letters.

Hint 3

One complete final assembly has this structure:

```python
vault_letters = []
for code_position, code_letter in sorted_clues:
    vault_letters.append(code_letter)

vault_word = "".join(vault_letters)
vault_status = ("SEALED", "OPEN")[vault_ready]

artifact = (
    "=== WAYFINDER SIGNAL VAULT ===\n"
    f"Status: {vault_status}\n"
    f"Code: {vault_word}\n"
    f"Signals: {len(active_ids)} active / {len(source_signals)} total\n"
    f"Sectors: {sector_text}\n"
    f"Symbols: {symbol_text}"
)
```

If the text still differs, compare `repr(artifact)` with `repr(expected_artifact)` to expose hidden spaces or newlines.

7. Keep debugging evidence

Preserve one real failed assertion or exception. A useful record connects the wrong collection decision to the observed value rather than saying only “the code did not work.”

Failure	Exact evidence	One hypothesis	Controlled change	Verified rerun
What was the first blocked check?	Include actual/expected values or the final exception line.	Which key, position, mutation, or collection operation explains it?	What one line changed?	Which unchanged group now passes?

Good puzzle evidence could include:

- appending a symbol set and creating a nested list instead of updating a set;
- counting only active signals instead of all sector records;
- directly traversing a dictionary and expecting complete records;
- applying only the strength sort and losing the explicit tie policy;
- unpacking clue fields in the wrong order; or
- mutating a source symbol set while combining membership.

8. Test the echo-signal variation

After the ordinary challenge passes from a clean state, append this record to a new outer list—not to `source_signals`:

```python
variation_signals = source_signals.copy()
variation_signals.append(
    {
        "id": "SIG-ECHO",
        "sector": "Vela",
        "strength": 70,
        "status": "inactive",
        "clue": (4, "?"),
        "symbols": {"star", "echo"},
```

PYTHON (CONTINUED)

```
    }
)
```

Copy your implementation to a new section, use `variation_signals` as its input, and recalculate every derived collection from empty state. Verify:

PYTHON

```
assert source_signals == source_snapshot
assert len(variation_signals) == 6
assert "SIG-ECHO" in signals_by_id
assert active_ids == ["SIG-N", "SIG-O", "SIG-A", "SIG-V"]
assert active_symbols == {"key", "moon", "star", "sun"}
assert sector_counts == {"Lyra": 2, "Orion": 2, "Vela": 2}
assert sorted_clues == [(0, "N"), (1, "O"), (2, "V"), (3, "A")]
assert vault_word == "NO" + "VA"
assert ranked_ids[-1] == "SIG-ECHO"
```

Explain why the lookup, total record count, Vela count, and ranking change while the active order, required-symbol readiness, clue order, and decoded word do not.

9. Compare with a solution path

Reveal after your checks pass or all three hints have been used

PYTHON

```
from operator import itemgetter

# Use source_signals, source_snapshot, and required_symbols from the starter.
signals_by_id = {}
active_ids = []
active_symbols = set()
sector_counts = {}
clue_pairs = []

for signal in source_signals:
    signal_id = signal["id"]
    sector = signal["sector"]

    signals_by_id[signal_id] = signal
    sector_counts[sector] = sector_counts.get(sector, 0) + 1

    if signal["status"] == "active":
        active_ids.append(signal_id)
        active_symbols.update(signal["symbols"])

        code_position, code_letter = signal["clue"]
        clue_pairs.append((code_position, code_letter))

missing_symbols = required_symbols - active_symbols
vault_ready = required_symbols <= active_symbols

ranked_signals = sorted(source_signals, key=itemgetter("id"))
ranked_signals = sorted(
    ranked_signals,
    key=itemgetter("strength"),
    reverse=True,
)

ranked_ids = []
for signal in ranked_signals:
    ranked_ids.append(signal["id"])

sorted_clues = sorted(clue_pairs)
```

PYTHON (CONTINUED)

```
vault_letters = []
for code_position, code_letter in sorted_clues:
    vault_letters.append(code_letter)

vault_word = "".join(vault_letters)
vault_status = ("SEALED", "OPEN")[vault_ready]

sector_text = (
    f"Lyra={sector_counts['Lyra']}, "
    f"Orion={sector_counts['Orion']}, "
    f"Vela={sector_counts['Vela']}"
)
symbol_text = ", ".join(sorted(active_symbols))

artifact = (
    "=== WAYFINDER SIGNAL VAULT ===\n"
    f"Status: {vault_status}\n"
    f"Code: {vault_word}\n"
    f"Signals: {len(active_ids)} active / {len(source_signals)} total\n"
    f"Sectors: {sector_text}\n"
    f"Symbols: {symbol_text}"
)

print(artifact)
```

The source list provides authoritative order. The lookup and counts include all records; the active list, symbol set, and clue tuples include only active records. Two stable ranking passes preserve an explicit ID tie rule. The word is derived from sorted fixed-position clues, and display sorting is applied only to a new symbol list.

10. Check your understanding

Answer these questions before recording the challenge. The quiz runs directly in your browser.

Signal-vault collection decisions

Choose the operation that preserves the archive contract.

1. Which view directly retrieves a complete signal from its stable ID?

- **A.** A dictionary mapping ID to record
- **B.** A set of strengths
- **C.** A sorted symbol list
- **D.** A tuple of sector counts

2. Which operation combines members of one signal's symbol set into `active_symbols`?

- **A.** `active_symbols.append(signal['symbols'])`
- **B.** `active_symbols.update(signal['symbols'])`
- **C.** `active_symbols[0] = signal['symbols']`
- **D.** `active_symbols.pop(signal['symbols'])`

3. Why does the ranking use `sorted()` rather than `source_signals.sort()`?

- **A.** Dictionaries cannot be sorted.
- **B.** sorted() removes inactive records automatically.

- **C.** It returns a ranked list while preserving source transmission order.
- **D.** sort() can compare only strings.

4. Why are active clue tuples sorted before their letters are joined?

- **A.** Sets require tuple sorting.
- **B.** Joining mutates tuples.
- **C.** Signal strength is stored in the tuple.
- **D.** The first tuple value is the intended code position, which differs from transmission order.

Write your choices before checking the answer key at the end of this book.

11. Decide whether the challenge is complete

Evidence rubric

Evidence	Ready to record when
Behavior	All twenty progressive checks pass unchanged and the artifact displays exactly.
Collection design	You can explain why each derived view is a list, tuple, dictionary, set, or `frozenset`.
Source preservation	The source snapshot passes before and after ranking and variation work.
Debugging	One failure record connects exact evidence, one hypothesis, one change, and a passing rerun.
Reproducibility	The ordinary challenge and variation work from separately initialized clean state.

Record completion only when every statement is true:

- ☐ All supplied assertions pass without editing their expected values.
- ☐ The artifact is derived rather than hard-coded.
- ☐ The work runs from a clean notebook state.
- ☐ I can explain each collection choice and mutation boundary in my own words.
- ☐ I kept one complete debugging record.
- ☐ I predicted and verified the echo-signal variation.

This button stores a self-reported marker only in this browser. It does not submit the artifact, grade it, verify identity, or issue a certificate.

Key points

Key points

- One ordered source can support keyed, grouped, unique, ranked, and display views when every view has a named purpose.

- Progressive assertions reveal the first broken collection contract before a plausible final string hides it.
- Stable sorting and complete tuple/record movement preserve relationships among tied or repositioned values.
- Sets answer unique-membership questions; frozen sets can state immutable requirements; lists preserve the orders the puzzle actually needs.
- A clean rerun, boundary variation, and debugging record are part of the solved artifact. :::

Unit 4: Making Decisions and Repeating Work

Online lesson and runnable notebook

Choose a control-flow tool from the question a program must answer, then build complete branches and loops that make visible progress.

- **Level:** Python Foundations · Unit 4
- **Estimated time:** 45–75 minutes for orientation and readiness work
- **You will learn:** Choose between a branch, a `for` loop, a `while` loop, early exit, nested traversal, and a comprehension from the behavior you need.
- **Practice in:** Google Colab, JupyterLab, or a local editor

1. One program uses three kinds of control

A supply drone receives a destination, checks whether launch is safe, and visits an ordered route. Those are different control-flow jobs:

```python
weather = "clear"
route = ["hangar", "ridge", "clinic"]
battery = 72

if weather == "clear" and battery >= 50:
    launch_status = "approved"
else:
    launch_status = "delayed"

for stop in route:
    print("Checking", stop)

while battery > 60:
    battery -= 5
    print("Calibration pulse; battery:", battery)
```

- The `if` statement chooses one path from the current facts.
- The `for` loop performs work once for each route item.
- The `while` loop repeats while a changing condition remains true.

Control flow is not only punctuation. It is the order in which Python evaluates questions, enters or skips blocks, changes state, and eventually continues after the statement. In this unit you will make every one of those steps visible.

2. Choose the tool from the question

Question the program must answer	Tool to consider	Correctness question
Which one of several results applies?	`if` / `elif` / `else`	Are the cases complete, and does order resolve overlaps deliberately?
What should happen to every supplied item?	`for`	Is the result initialized once and updated for every relevant item?
What should repeat until state changes?	`while`	What moves toward the stopping condition on every possible path?
Can the search stop after one result?	`break` or a direct membership operation	Do you need the first match, every match, or only a yes/no fact?
Should one unusable item be ignored?	`continue` or an inverted condition	Will later work and necessary progress still happen?
Does every row combine with every column?	nested loops	Is the resulting amount of work intentional?
Is a new collection described by one short rule?	comprehension	Is the compact form clearer than the equivalent loop?

The same data can lead to different choices. A list of temperatures suggests a `for` loop if every reading must be converted, `break` if one dangerous reading is enough to trigger an alarm, and a comprehension if the whole result is a small filtered list.

3. Trace the path, not only the output

Two programs can print the same result but reach it through different paths. Keep a small trace whenever the behavior is not obvious:

Moment	Current input	Question checked	State before	Action	State after
1	`"hangar"`	—	`visited = []`	append stop	`["hangar"]`
2	`"ridge"`	blocked?	one visit	skip or append	depends on answer
3	`"clinic"`	—	prior visits	append stop	final route

For a conditional, record the conditions in their evaluated order and mark the first selected branch. For a `for` loop, record the supplied item and changing result. For a `while` loop, also record why another iteration is or is not allowed. This turns “the code seems stuck” into evidence about one precise state transition.

4. Your path through this unit

Lesson	Capability you will build	Main lab artifact
1	Choosing a Path with `if`, `elif`, and `else` — distinguish a branch chain from independent decisions and ensure a usable result exists.	Spaceport departure gate
2	Combining Conditions Without Surprises — name facts, group rules, protect unsafe work, and preserve meaningful falsy values.	Museum security console
3	Turning Rules into Complete Decisions — expose gaps, overlaps, precedence, and exact boundaries before writing code.	Festival ticket desk
4	Building Results with `for` Loops — transform, filter, total, count, group, and validate without losing source data.	Expedition report
5	Repeating Until Something Changes — design progress, retry budgets, sentinels, and explicit exit reasons.	Docking controller
6	Searching, Skipping, and Stopping Loops — choose first, last, all, or existence results and explain loop `else`.	Signal scanner
7	Working Through Grids and Pairs — traverse coordinates, ragged rows, and combinations while controlling inner-loop exits.	Treasure-grid survey
8	Building Collections with Comprehensions — translate between explicit loops and readable list, set, or dictionary comprehensions.	Event-log cleanup

The Unit Challenge combines these capabilities in a clockwork maze. You will guide a maintenance robot through a grid while keeping an event log and proving each stage with assertions.

5. What Units 2 and 3 already supplied

Bring these skills forward rather than starting again:

- Unit 2: comparisons, truthiness, `and`, `or`, `not`, precedence, short-circuit evaluation, `None`, strings, and formatted output;
- Unit 3: lists, tuples, dictionaries, sets, nested data, range objects, `enumerate()`, `zip()`, sorting, membership, and safe traversal.

This unit asks a new question: **how should execution respond to those values?** You will apply Boolean facts to complete policies and collection operations to small algorithms. Unit 5 then gives those algorithms reusable names, parameters, and return values. The examples here remain top-level code so you can focus on control flow before learning function contracts.

6. Set up a control-flow laboratory

Use one cell for inputs, one for the algorithm, and one for checks. Temporary trace rows belong beside the code while you investigate:

```python
readings = [3, -1, 5]
accepted = []
rejected = []

for reading in readings:
    before = (accepted.copy(), rejected.copy())
    if reading >= 0:
        accepted.append(reading)
        action = "accepted"
    else:
        rejected.append(reading)
        action = "rejected"
    print(reading, before, action, accepted, rejected)

assert accepted == [3, 5]
assert rejected == [-1]
assert readings == [3, -1, 5]
```

After the checks pass, remove the temporary `print()` if it no longer helps. The assertions remain as executable promises. Restart and run all before considering an exercise finished; loops can otherwise appear correct only because an old notebook cell left useful state behind.

A loop can run correctly and still solve the wrong job

Visiting all items proves only that traversal happened. State the required artifact—such as “a new list of accepted readings in source order”—and assert that artifact, the empty case, and any important boundary. :::

7. Choose a realistic pace

Unit 4 is planned for approximately **27–40 hours**, including examples, modifications, checkpoints, eight final labs, and the challenge. The range is a guide, not a speed target.

A useful rhythm is:

1. trace a complete example before changing it;
2. change one boundary or input shape and update the prediction;
3. answer a checkpoint without rerunning the earlier code;
4. build the final lab from its contract before opening support; and
5. retain one failure with the state that caused it and the check that guards the repair.

8. Check your readiness

Run this short task from a clean cell:

```python
energy_readings = [8, 0, -2, 5]
usable = []

for reading in energy_readings:
    if reading < 0:
        continue
    usable.append(reading)

assert usable == [8, 0, 5]
assert energy_readings == [8, 0, -2, 5]
```

Then make three controlled changes:

1. collect negative readings in a separate `rejected` list instead of silently losing them;
2. add a `usable_total` accumulator and prove that it equals `13`; and
3. explain why the zero reading belongs in `usable` even though `bool(0)` is `False`.

You are ready for Lesson 1 when you can identify what the loop supplies, which branch runs for every input, which names change, and what remains unchanged.

Key points

Key points

- A branch chooses from current facts; a `for` loop consumes supplied items; a `while` loop repeats while changing state permits it.
- Trace conditions, supplied values, state changes, and the reason execution leaves a loop.
- Reuse Boolean and collection knowledge from Units 2 and 3 instead of hiding it inside unexplained control flow.
- Define and assert the result the program must produce, including boundaries and empty inputs. :::

Choosing a Path with `if`, `elif`, and `else`

Online lesson and runnable notebook

Build conditional branches that choose deliberately, cover a fallback, preserve useful results, and distinguish one exclusive decision from several independent checks.

- **Level:** Python Foundations · Unit 4
- **Estimated time:** 3–5 hours
- **You will learn:** Choose among paths with `if`, `elif`, and `else`, order branches safely, and ensure later code has a defined result.
- **Practice in:** Google Colab, JupyterLab, or a local editor

1. Decide whether a ship can depart

A spaceport gate should approve a departure only when its current facts allow it:

```python
fuel = 74
weather = "clear"
permit_active = True

if fuel >= 60 and weather == "clear" and permit_active:
    departure = "approved"
else:
    departure = "hold"

print(departure)
```

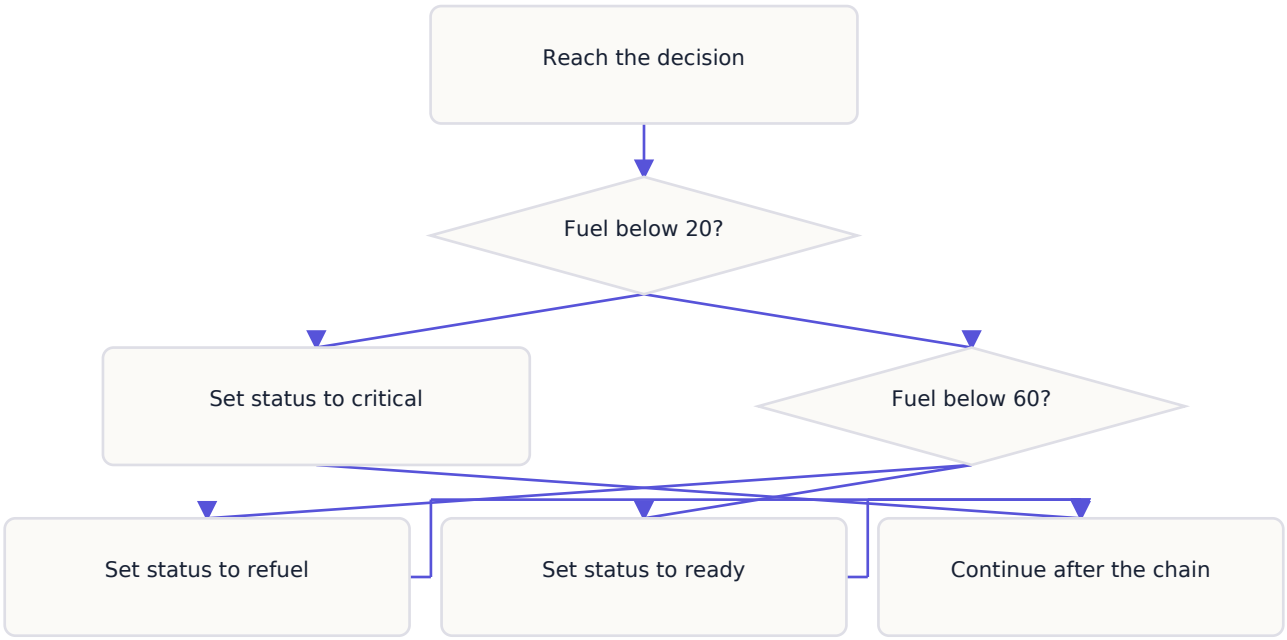
Python evaluates the condition after `if`. Because it is `True`, Python runs the first indented block and skips the `else` block. The name `departure` refers to `"approved"` afterward.

A conditional changes **which statements execute**. It does not make the facts true, repeat the block, or automatically validate every possible input. Your job is to state the rule and provide a useful path for each expected case.

Questions this lesson will answer

- How do indentation and colons mark a branch?
- When does an `elif` condition get evaluated?
- When should several checks be independent `if` statements?
- Why can branch order make a valid rule unreachable?
- How can later code rely on a result from the decision?

Python tests a branch chain from top to bottom and runs at most one branch.



Fuel below 20? → Set status to critical: Yes

Fuel below 20? → Fuel below 60?: No

Fuel below 60? → Set status to refuel: Yes

Fuel below 60? → Set status to ready: No

2. A colon opens a block and indentation owns its statements

A one-branch conditional runs its body only when the condition is true:

```python
temperature = 2

if temperature <= 3:
    print("Protect the seedlings")

print("Weather check complete")
```

The first `print()` belongs to the `if` because it is indented. The second is aligned with `if`, so it runs after the decision whether the condition was true or false.

The four visible parts are:

1. the `if` keyword;
2. an expression whose truth is tested;
3. a colon; and
4. a consistently indented body.

Omitting the colon is a syntax error. Removing the indentation is also a syntax error because Python expects a block after the header.

Indentation changes behavior

Do not indent a later statement merely because it looks related in English. Its indentation decides whether it belongs to the branch.

```python
is_raining = False

if is_raining:
    print("Take a coat")
    print("This line runs only when it is raining")

print("This line always runs")
```

:::

3. `else` supplies the remaining path

`else` does not have its own condition. It means “none of the earlier branches in this chain ran.”

```python
has_pass = False

if has_pass:
    gate_message = "Welcome aboard"
else:
    gate_message = "Pass required"

print(gate_message)
```

Exactly one assignment runs, so `gate_message` exists afterward for both cases. That property matters whenever later code needs a result.

Contrast it with this incomplete decision:

```python
has_pass = False

if has_pass:
    gate_message = "Welcome aboard"

print(gate_message)
```

The assignment is skipped, then `print()` tries to use a name that was never bound. Python raises `NameError`. Either assign a default before the decision or use an `else` when the two paths are the complete policy.

```python
gate_message = "Pass required"

if has_pass:
    gate_message = "Welcome aboard"
```

Both designs can be valid. `if`/`else` emphasizes two alternatives; a default followed by an override can be clearer when most inputs share the default.

Checkpoint: one or two paths

Branch structure checkpoint

Choose the behavior guaranteed by the code.

1. A print statement is aligned with `if` rather than its body. When does it run?

- **A.** After the conditional, regardless of which branch ran
- **B.** Only when the condition is true
- **C.** Only when the condition is false
- **D.** Before Python checks the condition

2. When does an `else` body run?

- **A.** Whenever its own hidden condition is true
- **B.** When no earlier branch in the same chain ran
- **C.** After every `if` body
- **D.** Only when a variable is `None`

3. Why can assigning `message` only inside an `if` cause `NameError` later?

- **A.** Strings cannot be assigned in blocks
- **B.** Python deletes every branch variable
- **C.** The condition may be false, so the assignment may never execute
- **D.** print() cannot read assigned names

4. What belongs at the end of an `if condition` header?

- **A.** A semicolon
- **B.** An equals sign
- **C.** A comma
- **D.** A colon

Write your choices before checking the answer key at the end of this book.

4. Only the first true branch in a chain runs

Use `elif` when one decision has more than two mutually exclusive outcomes:

PYTHON
```
fuel = 42

if fuel < 20:
    fuel_status = "critical"
elif fuel < 60:
    fuel_status = "refuel"
else:
    fuel_status = "ready"

print(fuel_status)
```

Python asks `fuel < 20` first. It is false, so Python evaluates `fuel < 60`. That is true, so Python assigns `"refuel"` and skips the rest of the chain. The `else` means `fuel` is neither below 20 nor below 60—in other words, it is at least 60.

A useful trace is:

Condition in order	Result for `fuel = 42`	Action
`fuel < 20`	`False`	continue down chain
`fuel < 60`	`True`	assign `"refuel"` and leave chain
`else`	not reached	skip

Test exact boundaries, not only typical values:

```python
for fuel in [19, 20, 59, 60]:
    if fuel < 20:
        status = "critical"
    elif fuel < 60:
        status = "refuel"
    else:
        status = "ready"
    print(fuel, status)
```

`20` belongs to `"refuel"`; `60` belongs to `"ready"`. Changing `<` to `<=` would move those endpoints. The operator is part of the policy.

5. Put specific cases before broader cases

Branch order resolves overlaps. Consider this reversed policy:

```python
score = 96

if score >= 70:
    result = "pass"
elif score >= 90:
    result = "distinction"
else:
    result = "retry"
```

Both comparisons are true for 96, but the first one wins. The distinction branch is unreachable. Put the more specific high threshold first:

```python
if score >= 90:
    result = "distinction"
elif score >= 70:
    result = "pass"
else:
    result = "retry"
```

This is not a Python preference for large values. It follows only from top-to- bottom, first-true evaluation. For low thresholds, the corresponding safe order may go from smallest to largest, as the fuel example did.

Read an `elif` using what already failed

In `elif fuel < 60`, the program has already learned that `fuel < 20` is false. The effective case is therefore `20 <= fuel < 60`, even though the second line does not repeat the lower bound. :::

6. Independent `if` statements answer independent questions

An `if`/`elif` chain runs at most one branch. Separate `if` statements may both run:

```python
energy = 18
alerts = []

if energy < 25:
    alerts.append("low energy")

if energy % 2 == 0:
    alerts.append("even calibration value")

print(alerts)
```

Both facts are true, and both alerts are useful. Replacing the second `if` with `elif` would suppress the calibration alert whenever energy is low.

Choose from the relationship between outcomes:

- **one classification:** use an `if`/`elif`/`else` chain;
- **several facts can coexist:** use independent `if` statements;
- **one result with a special override:** consider a default followed by one or more independent override checks, making precedence explicit.

Checkpoint: branch order and independence

Branch choice checkpoint

Trace which conditions are reached and how many outcomes may apply.

1. In an `if`/`elif`/`else` chain, what happens after one condition is true?

- **A.** Its body runs and the remaining chain is skipped
- **B.** Every later condition is still checked
- **C.** The first body is undone if `else` exists
- **D.** Python restarts the chain

2. Two alerts may both be relevant for one input. Which structure expresses that?

- **A.** One `if` followed by `elif`
- **B.** Two independent `if` statements
- **C.** An `else` with no `if`
- **D.** A conditional expression that discards one alert

3. Why is `elif score >= 90` unreachable after `if score >= 70`?

- **A.** Python cannot compare scores twice
- **B.** elif supports only smaller numbers
- **C.** Every score at least 90 already satisfies the first broader condition
- **D.** The else branch blocks it

4. With `if fuel < 20`, `elif fuel < 60`, `else`, which branch receives `fuel = 60`?

- **A.** The first branch
- **B.** The elif branch
- **C.** No branch
- **D.** The else branch

Write your choices before checking the answer key at the end of this book.

7. Nest only when the second question depends on the first

A nested conditional can reveal dependency:

```python
has_ticket = True
bag_weight = 14

if has_ticket:
    if bag_weight <= 20:
        boarding = "ready"
    else:
        boarding = "check bag"
else:
    boarding = "ticket required"
```

The weight question matters only after the ticket check succeeds. A compound condition is shorter when only the success case matters:

```python
if has_ticket and bag_weight <= 20:
    boarding = "ready"
else:
    boarding = "not ready"
```

But the shorter version collapses two useful failure explanations. Choose the shape from the result you need, not from line count.

Avoid a deep staircase of nested branches. When cases form one classification, a flat `elif` chain is usually easier to scan. When decisions genuinely depend on earlier answers, one or two levels can make the dependency clearer.

8. Common conditions read naturally

Membership can replace repeated equality tests:

```python
command = "launch"

if command in {"launch", "depart", "go"}:
    action = "start engines"
else:
    action = "wait"
```

Use `is None` for an intentionally missing value:

```python
assigned_dock = None

if assigned_dock is None:
    message = "Await dock assignment"
else:
    message = f"Proceed to dock {assigned_dock}"
```

Truthiness is useful only when every falsy value means the same thing:

```python
crew_names = []

if crew_names:
    manifest_status = "crew listed"
else:
    manifest_status = "manifest empty"
```

Do not use `if assigned_dock:` if dock `0` is a valid dock. That would confuse a valid falsy integer with missing information. Unit 2's truthiness rules still apply inside every branch.

9. A conditional expression is for one small value choice

Python can choose between two expressions in one line:

```python
is_night = True
lamp_mode = "on" if is_night else "off"
```

Read it as: assign `"on"` when `is_night` is true, otherwise assign `"off"`. Use this form for one short, readable value choice. Prefer a normal statement when either path has several actions, needs comments, or becomes hard to say aloud. Nested conditional expressions are legal but rarely kind to a beginner or future maintainer.

Checkpoint: expressing the rule clearly

Rule shape checkpoint

Choose the form that preserves the intended meaning.

1. Which condition clearly accepts `launch`, `depart`, or `go`?

- **A.** `command in {'launch', 'depart', 'go'}`
- **B.** `command == {'launch', 'depart', 'go'}`
- **C.** `command is True`
- **D.** `command and 'launch'`

2. Dock number 0 is valid and `None` means unassigned. Which check detects only unassigned?

- **A.** `if not dock:`
- **B.** `if dock is None:`
- **C.** `if dock == 0:`
- **D.** `if bool(dock):`

3. When can a nested conditional be clearer than one compound condition?

- **A.** Whenever the file is long
- **B.** Whenever both conditions are Boolean
- **C.** When the second question applies only after the first and distinct failure results matter
- **D.** When indentation should be avoided

4. Which task best suits a conditional expression?

- **A.** Run three statements on each path
- **B.** Classify five score ranges
- **C.** Repeat until input changes
- **D.** Choose one short value from two alternatives

Write your choices before checking the answer key at the end of this book.

10. Build a spaceport departure gate

Complete the lab from the contract before opening the support. The gate assigns one status and one explanation for every expected input.

```python
fuel = 58
weather = "clear"
permit_active = True
medical_priority = False

status = None
reason = None
```

Apply these rules in the stated precedence:

1. A medical-priority flight with an active permit is `"priority launch"`, even when fuel is below the normal threshold, but it still cannot launch in a storm.
2. A missing permit produces `"hold"` and reason `"permit inactive"`.
3. Storm weather produces `"hold"` and reason `"unsafe weather"`.
4. Fuel below 60 produces `"refuel"` and reason `"fuel below 60"`.
5. Every remaining flight is `"approved"` with reason `"all checks passed"`.
6. Add the independent alert `"reserve fuel"` when fuel is from 50 through 64, inclusive. It can coexist with any status.

After writing the decision, run:

```python
assert status == "refuel"
assert reason == "fuel below 60"
assert fuel_alert == "reserve fuel"

assert status in {"priority launch", "hold", "refuel", "approved"}
assert reason is not None
```

Then rerun from clean inputs for these cases and record the expected pair before you execute:

- `fuel = 10`, clear weather, active permit, medical priority;
- `fuel = 80`, storm, active permit, medical priority;
- `fuel = 80`, clear weather, inactive permit, no priority; and
- `fuel = 64`, clear weather, active permit, no priority.

Hint: separate classification from the independent alert

Use one `if`/`elif`/`else` chain for `status` and `reason`. Put the priority case before the ordinary restrictions, but include all facts that priority still requires. Use a separate `if`/`else` for `fuel_alert` because it is not another classification status.

Show one complete solution after attempting the lab

PYTHON

```
fuel = 58
weather = "clear"
permit_active = True
medical_priority = False

if medical_priority and permit_active and weather != "storm":
    status = "priority launch"
    reason = "medical priority"
elif not permit_active:
    status = "hold"
    reason = "permit inactive"
elif weather == "storm":
    status = "hold"
    reason = "unsafe weather"
elif fuel < 60:
    status = "refuel"
    reason = "fuel below 60"
else:
    status = "approved"
    reason = "all checks passed"

if 50 <= fuel <= 64:
    fuel_alert = "reserve fuel"
else:
    fuel_alert = "normal fuel band"
```

The classification chain selects exactly one status. The independent alert then answers a different question and can coexist with that result.

11. Explain the choices

Before leaving the lesson, answer in your own words:

1. How does alignment show which statements always run?
2. Why does an `else` often make a result name safe to use afterward?
3. What makes a broad condition dangerous before a more specific condition?
4. When are two independent `if` statements correct rather than an `elif` chain?
5. Why is `value is None` more precise than `not value` when zero is valid?

Key points

Key points

- Python evaluates an `if`/`elif` chain from top to bottom and runs only the first true branch, or `else` when no condition succeeds.
- Colons and indentation define branch ownership; alignment resumes execution after the statement.
- Put specific overlapping cases before broader ones and test exact boundaries.
- Use separate `if` statements when several outcomes may coexist.
- Ensure every path establishes the result later code needs.
- Use conditional expressions only for small two-way value choices. :::

References

- Python tutorial: `if` statements
- Python language reference: the `if` statement
- Python standard types: truth-value testing
- Python expression reference: conditional expressions

Combining Conditions Without Surprises

Online lesson and runnable notebook

Turn several facts into readable Boolean rules, use short-circuit evaluation as a safety guard, and avoid collapsing meaningful falsy states.

- **Level:** Python Foundations · Unit 4
- **Estimated time:** 3–4.5 hours
- **You will learn:** Name and combine Boolean facts, group mixed rules, guard unsafe operations with short-circuiting, and preserve distinctions such as `None` versus zero.
- **Practice in:** Google Colab, JupyterLab, or a local editor

1. Name the facts before combining the rule

A museum security console unlocks an archive only for an authorized visitor when the room is open and no alarm is active. One dense condition can express that, but named facts let a reader inspect the reasoning:

PYTHON

```
badge = "researcher"
hour = 14
alarm_active = False

is_authorized = badge in {"researcher", "curator"}
is_open = 9 <= hour < 17
is_safe = not alarm_active
can_enter = is_authorized and is_open and is_safe

print(is_authorized, is_open, is_safe)
print("Unlock archive:", can_enter)
```

Each comparison or membership test creates a Boolean. The final rule combines facts whose names say what they mean. If entry is unexpectedly denied, the three printed values immediately narrow the cause.

Unit 2 established how comparisons, truthiness, `and`, `or`, and `not` work. Here you will use those rules to design control-flow questions that remain readable at boundaries and safe when some operations are not always valid.

Questions this lesson will answer

- How do truth tables expose a rule’s complete behavior?
- Where do parentheses matter when `and` and `or` appear together?
- How can evaluation order prevent an invalid lookup or index?
- Why do `None`, `0`, `""`, and `[]` sometimes need different outcomes?
- When do `all()` and `any()` make a collection of facts clearer?

2. Truth tables turn words into evidence

For two facts, enumerate every combination instead of reasoning from one happy case:

`is_authorized`	`is_open`	`is_authorized and is_open`	`is_authorized or is_open`
`False`	`False`	`False`	`False`
`False`	`True`	`False`	`True`
`True`	`False`	`False`	`True`
`True`	`True`	`True`	`True`

- `and` requires every combined requirement.
- `or` accepts at least one alternative.
- `not` reverses one truth value.

Translate policy words deliberately:

Policy wording	Typical Boolean shape
“both,” “all,” “must also”	`A and B`
“either,” “at least one”	`A or B`
“unless,” “not,” “no alarm”	often includes `not A`
“exactly one”	`(A and not B) or (B and not A)`

Do not assume everyday “or” means exactly one. Python’s `or` is inclusive: it is true when one or both sides are true.

```python
has_key = True
knows_code = True

print(has_key or knows_code)
print((has_key and not knows_code) or (knows_code and not has_key))
```

The first result is true; the second is false because both alternatives are present.

3. Parentheses make mixed rules visible

Python evaluates `not` before `and`, and `and` before `or`. Therefore:

```python
is_staff = False
has_ticket = True
is_open = False

allowed = is_staff or has_ticket and is_open
print(allowed)
```

means:

```python
allowed = is_staff or (has_ticket and is_open)
```

A staff member can enter even while closed. If the intended rule is “the room must be open, and the person must be staff or have a ticket,” write:

```python
allowed = (is_staff or has_ticket) and is_open
```

The parentheses are not decoration. They identify one grouped idea before it is combined with another. Even when precedence would produce the same answer, use parentheses when the policy has an obvious phrase boundary.

A quick review technique is to substitute Boolean values:

```python
print(False or True and False)
print((False or True) and False)
```

Both happen to be false here, so also test a case that distinguishes the shapes:

```python
print(True or False and False)    # True
print((True or False) and False)  # False
```

Checkpoint: translating policies

Boolean policy checkpoint

Choose the expression that matches the stated rule.

1. Entry requires a badge and an open room. Which expression matches?

- **A.** `has_badge and is_open`
- **B.** `has_badge or is_open`
- **C.** `not has_badge`
- **D.** `has_badge == is_open == False`

2. The room must be open, and the visitor must be staff or have a ticket. Which expression is clearest?

- **A.** `is_staff or has_ticket and is_open`
- **B.** `(is_staff or has_ticket) and is_open`
- **C.** `is_staff and has_ticket or is_open`
- **D.** `not is_open or is_staff`

3. When is Python’s `A or B` false?

- **A.** When both are true
- **B.** When exactly one is true
- **C.** Only when both are false

- **D.** Whenever A and B differ

4. Which shape means exactly one of A and B is true?

- **A.** `A and B`
- **B.** `A or B`
- **C.** `not A and not B`
- **D.** `(A and not B) or (B and not A)`

Write your choices before checking the answer key at the end of this book.

4. Short-circuiting can protect an unsafe lookup

Python evaluates `and` from left to right. If the left side is falsy, the whole `and` expression cannot become truthy, so Python skips the right side:

```
PYTHON
codes = []

has_primary_code = bool(codes) and codes[0] == "A7"
print(has_primary_code)
```

`bool(codes)` is false, so `codes[0]` is never evaluated. No `IndexError` occurs. Reverse the operands and the guard arrives too late:

```
PYTHON
codes = []
has_primary_code = codes[0] == "A7" and bool(codes)
```

Python attempts the unsafe index first and raises `IndexError`. A guard must be on the left of the operation it protects.

A dictionary example uses membership before indexing:

```
PYTHON
visitor = {"name": "Mina"}

has_active_pass = (
    "pass" in visitor
    and visitor["pass"] == "active"
)
print(has_active_pass)
```

The membership test is false, so the missing key lookup is skipped. This is a control-flow effect inside an expression: not every written operand necessarily runs.

A safety guard should prove the next operation is valid

`data and data[0]` protects an index because a truthy sequence has at least one item. `"pass" in visitor and visitor["pass"] == "active"` protects a dictionary lookup because membership proves the key exists. :::

5. `or` also stops when the outcome is known

For `or`, a truthy left operand already determines success, so Python skips the right side:

```python
has_master_key = True
entered_code = "wrong"

can_open = has_master_key or entered_code == "A7"
print(can_open)
```

The code comparison is not needed. Short-circuiting is observable when the right side would fail:

```python
manual_override = True
settings = {}

can_start = manual_override or settings["mode"] == "automatic"
print(can_start)
```

This runs because the lookup is skipped. That does not mean missing settings are always acceptable; it means the policy explicitly permits the override to make the other fact irrelevant.

Do not hide important work behind `and` or `or` merely to save lines. Use a normal conditional when the skipped action changes state, needs an explanation, or would surprise a reader.

6. `and` and `or` return operands, not forced Booleans

Unit 2 introduced this rule. It matters when a condition also constructs data:

```python
label = ""
fallback = "untitled exhibit"

display_label = label or fallback
print(display_label)
```

Because `label` is falsy, `or` returns the second operand string. If `label` were `"Meteor Map"`, it would be returned unchanged.

This defaulting pattern is safe only when **every** falsy first value means “absent.” It is wrong if `0` is meaningful:

```python
requested_floor = 0
floor = requested_floor or 1
print(floor)  # 1, although floor 0 was requested
```

Preserve the distinction explicitly:

```python
floor = 1 if requested_floor is None else requested_floor
assert floor == 0
```

When you need a real Boolean for state or output, use a comparison or `bool(...)` instead of relying on the returned operand.

Checkpoint: evaluation order and guards

Short-circuit checkpoint

Trace which operands Python evaluates.

1. Why does `bool(items) and items[0] == 'key'` avoid an error for an empty list?

- **A.** The falsy left operand makes Python skip the right operand
- **B.** Index zero exists in every list
- **C.** and catches IndexError
- **D.** bool() adds a placeholder item

2. Which expression safely tests a possibly missing `mode` key?

- **A.** `settings['mode'] == 'auto' and 'mode' in settings`
- **B.** `'mode' in settings and settings['mode'] == 'auto'`
- **C.** `settings['mode'] or 'mode' in settings`
- **D.** `bool('mode') and settings['mode']`

3. For `True or missing['key']`, which statement is correct?

- **A.** The dictionary is filled automatically
- **B.** The expression always raises KeyError
- **C.** The lookup is skipped because the left operand is truthy
- **D.** or evaluates from right to left

4. Why can `value or default` be wrong when zero is valid?

- **A.** or accepts only strings
- **B.** Zero raises a syntax error
- **C.** default must be zero too
- **D.** Zero is falsy, so the expression replaces it with the default

Write your choices before checking the answer key at the end of this book.

7. De Morgan’s laws help invert a complete rule

Suppose entry is allowed when a visitor is authorized **and** the alarm is not active:

```python
can_enter = is_authorized and not alarm_active
```

The denial condition is the negation of that whole expression:

```python
must_deny = not (is_authorized and not alarm_active)
```

De Morgan’s laws give an equivalent form:

```python
must_deny = (not is_authorized) or alarm_active
```

The two transformations are:

- `not (A and B)` is equivalent to `(not A) or (not B)`;

- `not (A or B)` is equivalent to `(not A) and (not B)`.

Notice that the connective changes as each fact is negated. Test all Boolean combinations when an inversion controls access, safety, payment, or another important policy.

PYTHON
```
for authorized in [False, True]:
    for alarm in [False, True]:
        original = not (authorized and not alarm)
        rewritten = (not authorized) or alarm
        assert original == rewritten
```

This small exhaustive check is possible because two Boolean facts have only four combinations.

8. Impossible ranges reveal a broken rule

Some conditions can never be true:

PYTHON
```
age = 20
impossible = age < 13 and age >= 18
print(impossible)
```

No one value can be below 13 and at least 18 simultaneously. Perhaps the intended condition was `age < 13 or age >= 18`, describing values outside 13 through 17.

Other rules are always true:

PYTHON
```
always_true = age >= 13 or age < 18
```

Every number satisfies at least one side, including the overlapping middle. Writing a few boundary values—12, 13, 17, and 18—usually reveals the problem. Lesson 3 expands this technique into decision tables.

9. `all()` and `any()` combine a collection of facts

When facts already exist as a collection, built-ins can communicate the rule:

PYTHON
```
checks = [True, True, False]

print(all(checks))
print(any(checks))
```

- `all(checks)` is true only if every item is truthy.
- `any(checks)` is true if at least one item is truthy.

They also short-circuit while visiting the collection: `all()` can stop at the first falsy item; `any()` can stop at the first truthy item.

The empty cases are deliberate:

PYTHON
```
assert all([]) is True
assert any([]) is False
```

“All zero checks passed” is true because no failing check exists. “At least one of zero checks passed” is false because no successful check exists. Whether an empty collection should be accepted by your application is a separate policy; you can require both non-emptiness and `all(checks)`:

```python
checks = []
ready = bool(checks) and all(checks)
assert ready is False
```

Checkpoint: complete Boolean reasoning

Complete rule checkpoint

Choose the result supported by Boolean identities and collection rules.

1. Which expression is equivalent to `not (A and B)`?

- **A.** `(not A) or (not B)`
- **B.** `(not A) and (not B)`
- **C.** `A or B`
- **D.** `A and not B`

2. What is true of `value < 10 and value >= 20` for ordinary numbers?

- **A.** It accepts values 10 through 20
- **B.** It is impossible because one value cannot satisfy both ranges
- **C.** It is true only at 20
- **D.** Python interprets it as or

3. What does `all([])` return?

- **A.** None
- **B.** An empty list
- **C.** True
- **D.** False

4. Which expression requires at least one check and requires every check to be truthy?

- **A.** `any(checks)`
- **B.** `all(checks)`
- **C.** `not checks or all(checks)`
- **D.** `bool(checks) and all(checks)`

Write your choices before checking the answer key at the end of this book.

10. Build a museum security console

Use these inputs and create named facts rather than one unreadable condition:

```python
visitor = {"role": "researcher", "pass": "active"}
hour = 16
alarm_active = False
restricted_exhibit = True
escort_present = True

access = None
reason = None
```

The policy is:

1. The museum is open from hour 9 through 16; hour 17 is closed.
2. The visitor is authorized when the role is `"curator"` or `"researcher"`.
3. A pass is valid only if the `"pass"` key exists and equals `"active"`.
4. No one enters while the alarm is active.
5. A restricted exhibit additionally requires either a curator or an escort.
6. Assign Boolean `access` and one specific `reason`. Check general failures in this order: closed museum, alarm, authorization/pass, restricted access.
7. Do not index the pass before proving its key exists.

Run these checks for the supplied case:

```python
assert is_open is True
assert is_authorized is True
assert has_active_pass is True
assert restriction_satisfied is True
assert access is True
assert reason == "access granted"
```

Then try a missing pass, `hour = 17`, an active alarm, and a researcher at the restricted exhibit without an escort. For each case, predict the first failing fact and the resulting reason.

Hint: build facts from simple to dependent

Create `is_open`, `is_authorized`, and `has_active_pass` separately. The pass fact should use membership on the left of `and`. Define the restricted-exhibit fact so that unrestricted exhibits pass without needing an escort. Use a branch chain to choose the most useful reason.

Show one complete solution after attempting the lab

```python
visitor = {"role": "researcher", "pass": "active"}
hour = 16
alarm_active = False
restricted_exhibit = True
escort_present = True

is_open = 9 <= hour < 17
is_authorized = visitor["role"] in {"curator", "researcher"}
has_active_pass = (
    "pass" in visitor
    and visitor["pass"] == "active"
)
restriction_satisfied = (
    not restricted_exhibit
```

PYTHON (CONTINUED)

```
    or visitor["role"] == "curator"
    or escort_present
)

if not is_open:
    access = False
    reason = "museum closed"
elif alarm_active:
    access = False
    reason = "alarm active"
elif not (is_authorized and has_active_pass):
    access = False
    reason = "authorization failed"
elif not restriction_satisfied:
    access = False
    reason = "escort required"
else:
    access = True
    reason = "access granted"
```

The branch chain reports one precedence-ordered reason. The named facts preserve which part of the policy each expression represents.

11. Explain the choices

1 Why are named Boolean facts easier to debug than one dense condition?

2 How can parentheses change a policy that mixes `and` and `or`?

3 What must a left-hand guard prove before Python reaches an unsafe right side?

4 Why is `value or default` unable to preserve every valid zero?

5 What do `all([])` and `any([])` mean, and when might a policy additionally require a non-empty input?

Key points

Key points

- Name meaningful facts, then combine those names into the policy.
- Parenthesize mixed `and`/`or` rules according to their logical phrases.
- `and` skips its right operand after a falsy left operand; `or` skips it after a truthy left operand. Put safety guards before the operations they protect.
- Truthiness can collapse meaningful states such as zero and `None`; compare explicitly when the distinction matters.
- De Morgan’s laws invert a whole rule, and boundary cases reveal impossible or always-true ranges.
- `all()` and `any()` combine collections of facts and have deliberate empty behavior. :::

References

- Python standard types: Boolean operations
- Python standard types: truth-value testing

- Python expression reference: evaluation order
- Python built-in function: `all`
- Python built-in function: `any`

Turning Rules into Complete Decisions

Online lesson and runnable notebook

Convert prose policies into decision tables, find boundary gaps and overlaps, encode precedence, and use simple match statements for discrete choices.

- **Level:** Python Foundations · Unit 4
- **Estimated time:** 3–5 hours
- **You will learn:** Design complete, non-overlapping decisions from tables, test exact boundaries, apply precedence, and recognize when `match` clarifies a discrete command.
- **Practice in:** Google Colab, JupyterLab, or a local editor

1. Turn a ticket policy into rows before code

A night festival publishes this policy:

- children under 13 pay 8 credits;
- visitors from 13 through 64 pay 18 credits;
- visitors 65 or older pay 10 credits; and
- volunteers with an active badge enter free.

The words sound clear, but code forces exact answers. A **decision table** makes the cases and their precedence visible first:

Priority	Facts	Price	Reason
1	active volunteer badge	0	volunteer
2	age below 13	8	child
3	age from 13 through 64	18	standard
4	age at least 65	10	senior

The volunteer row is an override, so it appears first. The age rows are mutually exclusive and together cover every expected non-negative age.

PYTHON

```
age = 12
active_volunteer = False

if active_volunteer:
    price = 0
    reason = "volunteer"
elif age < 13:
    price = 8
    reason = "child"
elif age < 65:
    price = 18
```

PYTHON (CONTINUED)

```
    reason = "standard"
else:
    price = 10
    reason = "senior"

print(price, reason)
```

The second age condition can say only `age < 65` because reaching it already proves `age >= 13`. The table preserves the full intended interval.

Questions this lesson will answer

- How can a table reveal missing, overlapping, or unreachable cases?
- Which endpoints belong on each side of a threshold?
- How does precedence differ from an accidental branch-order effect?
- When should a result include an explanation as well as a value?
- When is `match` clearer than a chain of equality tests?

2. Separate mutually exclusive cases from cumulative rules

A single classification should select exactly one row. Ticket price is one classification: one visitor receives one base price. Use an `if`/`elif`/`else` chain.

Cumulative rules may all apply. A visitor could receive several notices:

PYTHON

```
notices = []
age = 70
active_volunteer = True

if age >= 65:
    notices.append("offer accessible seating")

if active_volunteer:
    notices.append("thank volunteer")

print(notices)
```

Both independent facts are useful, so two `if` statements are correct. A table can identify this distinction:

Rule type	Expected selected rows	Python shape
classification	exactly one	`if` / `elif` / `else`
optional override followed by classification	first matching precedence row	ordered branch chain
cumulative notices or effects	zero, one, or many	independent `if` statements

Mixing these purposes is a common source of missing work. Decide whether rows are alternatives or cumulative before translating them.

3. Boundaries belong to one case, not two or none

Suppose a temperature policy says cold is below 10 and warm is above 10:

```python
if temperature < 10:
    label = "cold"
elif temperature > 10:
    label = "warm"
```

At exactly 10, neither branch runs. If `label` was not already defined, later use raises `NameError`. This is a **gap**.

Now suppose the policy uses `temperature <= 10` and then `temperature >= 10`. Both conditions include 10. In an `elif` chain only the first wins, but the table still has an **overlap** whose outcome depends on row order.

Write interval notation in ordinary words:

Case	Included values	Boundary checks
cold	below 10	9
mild	10 through 19	10, 19
warm	20 or above	20

```python
if temperature < 10:
    label = "cold"
elif temperature < 20:
    label = "mild"
else:
    label = "warm"
```

Test one value just below, exactly at, and just above every threshold: 9, 10, 11, 19, 20, and 21. Typical middle values do not prove endpoint behavior.

Checkpoint: complete case design

Complete cases checkpoint

Identify the case relationship before choosing syntax.

1. Exactly one admission category must be assigned. Which structure best expresses that?

- **A.** One `if`/`elif`/`else` chain
- **B.** Several unrelated `while` loops
- **C.** Independent `if` statements with no fallback
- **D.** A set literal alone

2. A visitor may need both an accessibility notice and a volunteer notice. Which structure fits?

- **A.** An if/elif chain that stops after one notice
- **B.** Independent if statements

- **C.** One conditional expression
- **D.** An empty else branch

3. What is wrong with cases `value < 10` and `value > 10` if every number needs a result?

- **A.** They overlap for every value
- **B.** They accept only strings
- **C.** The exact value 10 belongs to neither case
- **D.** The first comparison is always true

4. Which values best test a threshold at 20?

- **A.** 1, 2, 3
- **B.** 20 only
- **C.** 0 and 100
- **D.** 19, 20, and 21

Write your choices before checking the answer key at the end of this book.

4. Precedence makes overrides deliberate

Some rows overlap because the policy says one rule overrides another. The volunteer can also be a child or senior, yet the free-price row wins. Record that priority rather than pretending the cases are disjoint.

A weather closure can override every ticket rule:

Priority	Condition	Result
1	festival closed by weather	unavailable
2	active volunteer	free
3	age band	band price

PYTHON

```
festival_open = False
active_volunteer = True
age = 35

if not festival_open:
    price = None
    reason = "festival closed"
elif active_volunteer:
    price = 0
    reason = "volunteer"
elif age < 13:
    price = 8
    reason = "child"
elif age < 65:
    price = 18
    reason = "standard"
else:
    price = 10
    reason = "senior"
```

`None` communicates that a price is unavailable, not zero. A zero price is a real price for a free admission. This distinction would disappear with a simple truthiness check.

A precedence question should be answerable in words: “weather closure wins over volunteer status, which wins over age.” If you can only explain it as “whatever branch happens to come first,” the design is not finished.

5. Keep a result and its reason together in the trace

A lone result can be hard to diagnose. Produce a reason in the same selected branch:

PYTHON
```
score = 84
submitted = True

if not submitted:
    outcome = "incomplete"
    reason = "submission missing"
elif score >= 90:
    outcome = "distinction"
    reason = "score at least 90"
elif score >= 70:
    outcome = "pass"
    reason = "score from 70 through 89"
else:
    outcome = "retry"
    reason = "score below 70"

print(outcome, "—", reason)
```

The two names should be assigned together on every path. A trace table can then record both:

Input	Conditions reached	Outcome	Reason
not submitted	first true	incomplete	submission missing
submitted, 90	second true	distinction	score at least 90
submitted, 70	third true	pass	score from 70 through 89
submitted, 69	fallback	retry	score below 70

That table doubles as a compact test plan.

6. Find overlaps mechanically with sample rows

When intervals are complex, enumerate representative values and count matching rules:

PYTHON
```
for value in [9, 10, 19, 20]:
    matches = 0
    matches += value <= 10
    matches += 10 <= value < 20
    matches += value >= 20
    print(value, matches)
```

Booleans behave like 1 and 0 in this narrow counting use. A classification row with `matches == 0` has a gap. A row with `matches > 1` has an overlap. Here the value 10 matches two conditions.

This diagnostic does not replace the final branch chain. It helps you inspect whether the written table matches the intended relationship.

Checkpoint: precedence and evidence

Precedence and evidence checkpoint

Choose the design that keeps overlapping rules understandable.

1. Weather closure must override every admission price. Where should that row appear?

- **A.** First in the precedence chain
- **B.** After the final else
- **C.** Inside every age comparison
- **D.** Its position cannot matter

2. Why use `None` for unavailable price and 0 for free admission?

- **A.** Python cannot print negative prices
- **B.** They represent different states: no price versus a real zero price
- **C.** None is mathematically equal to zero
- **D.** Only None can enter an if statement

3. What debugging benefit comes from assigning a reason beside the result?

- **A.** It makes conditions run twice
- **B.** It removes the need for boundary tests
- **C.** It records which policy fact selected the result
- **D.** It automatically catches exceptions

4. For mutually exclusive table rows, what does a sample with two matching rows reveal?

- **A.** A syntax error
- **B.** A missing fallback
- **C.** A correctly empty case
- **D.** An overlap

Write your choices before checking the answer key at the end of this book.

7. Use `match` for clear discrete choices

Python's `match` statement can make a menu of literal commands easy to scan:

```python
command = "map"

match command:
    case "map":
        response = "show route"
    case "scan" | "inspect":
        response = "scan nearby area"
    case "quit":
```

PYTHON (CONTINUED)

```
        response = "end session"
    case _:
        response = "unknown command"

print(response)
```

The cases are checked in order. The vertical bar means either literal pattern. The _ wildcard is the fallback and should come last because it matches anything.

For one or two equality checks, an `if` statement is often simpler. Use `match` when several **discrete shapes or commands** are the natural table. Do not use it to replace ordered numeric ranges:

PYTHON

```
if age < 13:
    price = 8
elif age < 65:
    price = 18
else:
    price = 10
```

This lesson uses only literal choices, alternatives, and the wildcard. More advanced structural patterns belong after you have reusable data models.

8. Avoid a wildcard that hides a required error state

A fallback needs a deliberate meaning. For an interactive command, `"unknown command"` is useful. For internal states that should already be validated, a wildcard that silently substitutes a normal result can hide a bug.

PYTHON

```
status = "mystery"

match status:
    case "ready":
        message = "launch"
    case "hold":
        message = "wait"
    case _:
        message = "invalid status"
```

The fallback keeps `message` defined but does not pretend the unknown state is ready. Unit 8 will introduce raising and handling exceptions. For now, preserve an unmistakable invalid result and assert the allowed states near their source.

Checkpoint: discrete choices

Discrete choices checkpoint

Choose when a simple match statement improves the decision.

1. Which task is a good fit for a simple `match` statement?

- **A.** Map several literal commands to responses
- **B.** Repeat while fuel changes
- **C.** Add every score

- **D.** Classify a continuous temperature range

2. What does `case 'scan' | 'inspect':` mean in this lesson?

- **A.** Both strings must be concatenated
- **B.** Either literal selects the same case
- **C.** One command is divided by another
- **D.** The case repeats twice

3. What does `case _:` provide when placed last?

- **A.** A numeric comparison
- **B.** A loop target
- **C.** A fallback for any unmatched value
- **D.** A comment

4. Which structure is clearer for ordered numeric bands such as below 13, below 65, otherwise?

- **A.** A dictionary with no conditions
- **B.** A while loop
- **C.** A wildcard-only match
- **D.** An ordered if/elif/else chain

Write your choices before checking the answer key at the end of this book.

9. Build the festival ticket desk

Create a complete decision for these inputs:

```python
festival_open = True
age = 67
active_volunteer = False
time_slot = "evening"

price = None
category = None
reason = None
```

Use this contract:

1. If the festival is closed, `price` stays `None`, category is `"unavailable"`, and the reason identifies closure.
2. Active volunteers enter free before any age rule.
3. Under 13 costs 8 credits; ages 13 through 64 cost 18; age 65 or above costs
4. A morning time slot subtracts 2 credits from any positive price. Free and unavailable admission do not change.
5. Keep the original category reason and add `"; morning discount"` only when the discount applies.
6. Produce `summary` containing the category, displayed price (`"N/A"` for `None`), and reason.

Run the supplied-case checks:

```python
assert price == 10
assert category == "senior"
assert reason == "age at least 65"
assert summary == "senior | 10 credits | age at least 65"
```

Create a table and test at least these cases: closed volunteer, ages 12, 13, 64, 65, a morning child, a morning senior, and a morning volunteer.

Hint: make classification and discount separate stages

Use one precedence-ordered branch chain to set `price`, `category`, and `reason` together. Then use an independent condition requiring `price is not None` and `price > 0` before subtracting the morning discount.

Show one complete solution after attempting the lab

```python
festival_open = True
age = 67
active_volunteer = False
time_slot = "evening"

if not festival_open:
    price = None
    category = "unavailable"
    reason = "festival closed"
elif active_volunteer:
    price = 0
    category = "volunteer"
    reason = "active volunteer"
elif age < 13:
    price = 8
    category = "child"
    reason = "age below 13"
elif age < 65:
    price = 18
    category = "standard"
    reason = "age from 13 through 64"
else:
    price = 10
    category = "senior"
    reason = "age at least 65"

if time_slot == "morning" and price is not None and price > 0:
    price -= 2
    reason += "; morning discount"

price_text = "N/A" if price is None else f"{price} credits"
summary = f"{category} | {price_text} | {reason}"
```

The second stage is cumulative policy, not another admission category. Explicit `is not None` preserves unavailable versus free.

10. Explain the table

1. Which rows are mutually exclusive, and which rule is cumulative?
2. Why does a volunteer override need a stated priority?
3. How do 12, 13, 64, and 65 expose every age endpoint?

4 Why are price and reason assigned in the same branch?

5 When is a literal `match` clearer than an `if` chain, and when is it not?

Key points

Key points

- Write facts, priorities, results, and reasons as table rows before translating a multi-rule policy.
- A classification selects one case; cumulative rules require independent checks.
- Test just below, at, and just above every threshold to find gaps and overlaps.
- State override precedence in words and encode the highest priority first.
- Preserve a reason beside a result so the selected path remains observable.
- Use simple `match` cases for discrete literals; use ordered comparisons for numeric ranges. :::

References

- Python tutorial: `if` statements
- Python tutorial: `match` statements
- Python language reference: the `match` statement
- Python standard types: comparisons

Building Results with `for` Loops

Online lesson and runnable notebook

Use for-loop accumulator patterns to total, count, transform, filter, group, validate, and collect unique values while preserving source evidence.

- **Level:** Python Foundations · Unit 4
- **Estimated time:** 4–5.5 hours
- **You will learn:** Initialize and update loop results for transforming, filtering, totaling, counting, grouping, validation, and unique collection.
- **Practice in:** Google Colab, JupyterLab, or a local editor

1. A loop needs a product, not only repeated output

Unit 3 showed how a `for` loop receives collection values. Most useful loops do more than print them: they build a result that remains after traversal.

```python
distances = [4, 7, 3]
total_distance = 0

for distance in distances:
    total_distance += distance

print(total_distance)
assert total_distance == 14
```

`total_distance` is an **accumulator**: state initialized before the loop and updated once for each relevant item. Its job can be stated precisely:

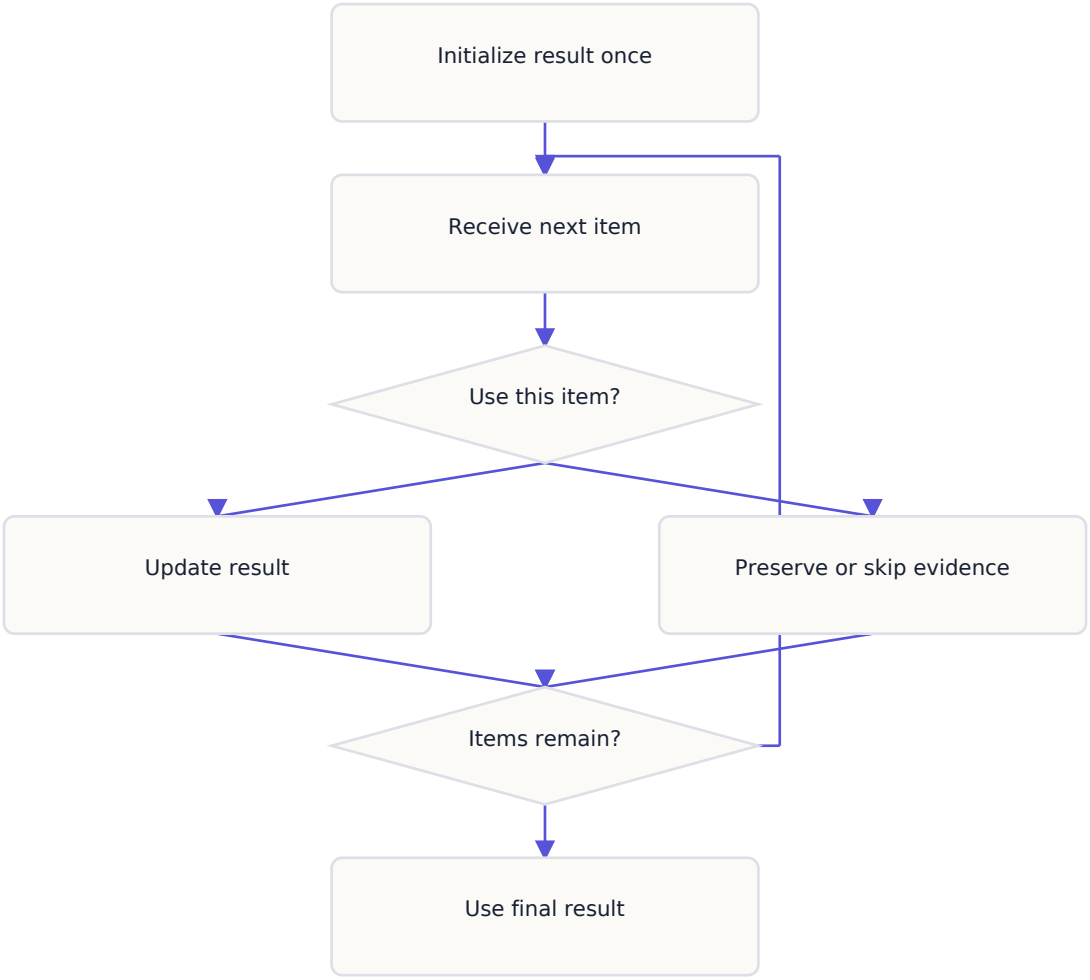
After each iteration, `total_distance` equals the sum of distances already visited.

That statement is a practical **loop invariant**. It helps you decide where to initialize the name, what each iteration must change, and what the final value means.

Questions this lesson will answer

- Which starting value fits a total, count, list, dictionary, or set result?
- How do transform and filter patterns differ?
- How can one traversal preserve both accepted and rejected evidence?
- How do counters and grouped records use dictionaries?
- When is a built-in clearer and safer than a manual loop?

A result-building loop starts with an empty or neutral result, receives one item, and updates the result before moving to the next item.



Use this item? → Update result: Yes

Use this item? → Preserve or skip evidence: No

Items remain? → Receive next item: Yes

Items remain? → Use final result: No

2. Initialize once, before traversal

Moving initialization inside the loop erases earlier work:

PYTHON

```
distances = [4, 7, 3]

for distance in distances:
    total_distance = 0
    total_distance += distance

print(total_distance)
```

The displayed value is 3, the final item, because each iteration resets the total. On an empty list, the body never runs and `total_distance` would not exist at all.

The correct pattern gives the result a meaningful empty-input value:

```python
total_distance = 0

for distance in []:
    total_distance += distance

assert total_distance == 0
```

Common initial values follow the promised result:

Result job	Initial value	Update example
numeric total	`0`	`total += value`
number of matches	`0`	`count += 1`
ordered transformed or filtered values	`[]`	`result.append(value)`
values grouped or counted by key	`{}`	update one key
unique observed values	`set()`	`seen.add(value)`
every item is valid	`True`	set false when one fails
at least one item matches	`False`	set true when one matches

An initial value is not boilerplate. It defines what the result means before any input arrives.

3. Total and count answer different questions

A total adds item values. A count adds one for each qualifying item:

```python
readings = [8, -2, 0, 5, -1]
usable_total = 0
usable_count = 0

for reading in readings:
    if reading >= 0:
        usable_total += reading
        usable_count += 1

print(usable_total, usable_count)
assert usable_total == 13
assert usable_count == 3
```

The zero reading contributes nothing to the total but still contributes one to the count. Using `if reading:` would incorrectly exclude it.

The average requires both accumulators and an empty-case policy:

```python
if usable_count > 0:
    usable_average = usable_total / usable_count
else:
    usable_average = None
```

`None` says no average exists. Returning zero would confuse “no readings” with “readings whose average is zero.”

Checkpoint: accumulator foundations

Accumulator foundations checkpoint

Choose the state that remains correct before and after every iteration.

1. Which initial value fits an ordered collection of accepted items?

- **A.** `[]`
- **B.** `0`
- **C.** `True`
- **D.** `None` with no later replacement

2. Why should a running total normally be initialized before the loop?

- **A.** Python forbids assignment in loops
- **B.** Each iteration must preserve the contributions from earlier items
- **C.** The target item is unavailable in the body
- **D.** It makes every item equal

3. For usable readings defined as `reading >= 0`, what does a zero reading contribute?

- **A.** One to the total and zero to the count
- **B.** Nothing to either value
- **C.** Zero to the total and one to the count
- **D.** It stops the loop

4. Why can `None` be a better empty-input average than 0?

- **A.** None can be divided safely
- **B.** Zero is not a number
- **C.** Lists cannot contain zero
- **D.** It distinguishes no average from an actual average of zero

Write your choices before checking the answer key at the end of this book.

4. Transform every item into a new value

A transformation produces one output for every input, usually preserving order:

```python
celsius_readings = [0, 10, 25]
fahrenheit_readings = []

for celsius in celsius_readings:
    fahrenheit = celsius * 9 / 5 + 32
    fahrenheit_readings.append(fahrenheit)

assert fahrenheit_readings == [32.0, 50.0, 77.0]
```

PYTHON (CONTINUED)

```
assert celsius_readings == [0, 10, 25]
```

The source remains evidence. Build a separate result rather than overwriting the input unless in-place mutation is an explicit requirement.

For richer records, preserve relationships:

PYTHON

```
stations = [
    {"name": "north", "reading": 3},
    {"name": "east", "reading": 7},
]
labels = []

for station in stations:
    label = f"{station['name']}: {station['reading']} units"
    labels.append(label)

assert labels == ["north: 3 units", "east: 7 units"]
```

The loop target `station` is rebound to each dictionary. The list accumulator keeps derived strings in source order.

5. Filter by appending only accepted items

A filter can produce zero or one output per input:

PYTHON

```
readings = [8, -2, 0, 5, -1]
usable = []

for reading in readings:
    if reading >= 0:
        usable.append(reading)

assert usable == [8, 0, 5]
```

The append belongs inside the conditional because acceptance decides whether the item enters the result. If it were aligned with `if`, every item would be added.

Silently dropping rejected data may make debugging harder. Preserve it when the program needs an audit trail:

PYTHON

```
usable = []
rejected = []

for reading in readings:
    if reading >= 0:
        usable.append(reading)
    else:
        rejected.append(reading)

assert usable == [8, 0, 5]
assert rejected == [-2, -1]
```

Now every source item belongs to exactly one result. A useful invariant is:

After each iteration, `len(usable) + len(rejected)` equals the number of items visited so far.

6. Combine transformation and filtering in an explicit order

Validate before applying an operation that assumes valid data:

```python
raw_scores = ["8", "skip", "10"]
clean_scores = []
rejected_tokens = []

for token in raw_scores:
    if token.isdigit():
        score = int(token)
        clean_scores.append(score * 10)
    else:
        rejected_tokens.append(token)

assert clean_scores == [80, 100]
assert rejected_tokens == ["skip"]
```

The branch protects `int(token)`. The transformation happens only on accepted text. This is a small data pipeline: inspect, classify, convert, store.

Unit 8 will teach exception handling for conversions whose valid forms are more complex. At this stage, choose inputs and guards that your current tools can explain.

Checkpoint: transform and filter

Transform and filter checkpoint

Trace how many outputs each input can create.

1. A loop converts every Celsius value and preserves order. How many outputs should each input create?

- **A.** Exactly one
- **B.** Always zero
- **C.** Two or more
- **D.** Only one output for the whole list

2. Where should `accepted.append(item)` appear in a filtering loop?

- **A.** Before result initialization
- **B.** Inside the branch whose condition defines acceptance
- **C.** After the loop with no item name
- **D.** Inside the rejection branch

3. What is a benefit of collecting rejected values separately?

- **A.** It mutates the source automatically
- **B.** It guarantees every value is valid
- **C.** It preserves evidence about what the filter excluded
- **D.** It eliminates conditional logic

4. Why check `token.isdigit()` before `int(token)` in this limited example?

- **A.** isdigit() converts the token
- **B.** int() works only in loops
- **C.** The check sorts the strings
- **D.** The check prevents this conversion from running on the known rejected form

Write your choices before checking the answer key at the end of this book.

7. Count occurrences with a dictionary

A counter connects each observed value with how many times it appeared:

```python
signals = ["blue", "red", "blue", "gold", "red", "blue"]
counts = {}

for signal in signals:
    if signal in counts:
        counts[signal] += 1
    else:
        counts[signal] = 1

assert counts == {"blue": 3, "red": 2, "gold": 1}
```

On the first occurrence, create the key with count 1. Later occurrences update the existing value. Dictionary `.get()` can express the same default:

```python
counts = {}

for signal in signals:
    counts[signal] = counts.get(signal, 0) + 1
```

Read the right side first: obtain the existing count or zero, add one, then store the new count at the same key.

8. Group complete records by a key

Grouping maps one key to several original records:

```python
observations = [
    {"zone": "north", "species": "owl"},
    {"zone": "south", "species": "fox"},
    {"zone": "north", "species": "moth"},
]
by_zone = {}

for observation in observations:
    zone = observation["zone"]
    if zone not in by_zone:
        by_zone[zone] = []
    by_zone[zone].append(observation)

assert [item["species"] for item in by_zone["north"]] == ["owl", "moth"]
```

The empty list must be created separately for each new key. Assigning one shared list to many keys would mix groups. Unit 6 develops aliasing in depth; for now, create the group exactly when its key first appears.

9. A set accumulator keeps unique observations

```python
species = ["owl", "fox", "owl", "moth"]
unique_species = set()

for name in species:
    unique_species.add(name)

assert unique_species == {"owl", "fox", "moth"}
```

The set answers uniqueness and membership questions. It does not preserve a numeric position. If you need unique values in first-seen order, combine a set for fast “seen?” checks with a list for ordered output:

```python
seen = set()
first_seen = []

for name in species:
    if name not in seen:
        seen.add(name)
        first_seen.append(name)

assert first_seen == ["owl", "fox", "moth"]
```

The two accumulators have different jobs and stay synchronized inside the same branch.

10. Validation accumulates a Boolean claim

To report whether every value is in range:

```python
levels = [4, 8, 11]
all_valid = True

for level in levels:
    if not 0 <= level <= 10:
        all_valid = False

assert all_valid is False
```

`all_valid` starts true because no visited item has disproved the claim. Once false, it stays false. If you also need invalid values, collect them instead and derive `all_valid = not invalid_levels` afterward.

A separate “at least one” claim starts false and becomes true when a match is seen. Lesson 6 shows how a search can stop early when no additional evidence is needed.

11. Use positions and pairs only when the result needs them

`enumerate()` supplies a position with each value:

```python
stops = ["dock", "ridge", "clinic"]
numbered = []

for number, stop in enumerate(stops, start=1):
    numbered.append(f"{number}. {stop}")

assert numbered == ["1. dock", "2. ridge", "3. clinic"]
```

`zip()` supplies aligned values from several collections and stops at the shortest:

```python
names = ["north", "east", "south"]
readings = [4, 7, 2]
station_readings = {}

for name, reading in zip(names, readings, strict=True):
    station_readings[name] = reading

assert station_readings == {"north": 4, "east": 7, "south": 2}
```

Unit 3 taught those supply mechanics. Here they serve a result-building job. `strict=True` raises `ValueError` when one input ends early, preventing silent loss when aligned records are required.

12. Prefer a built-in when it says the complete job

Python already names common reductions:

```python
values = [4, 7, 3]

assert sum(values) == 14
assert len(values) == 3
assert min(values) == 3
assert max(values) == 7
```

`sum(values)` is clearer than a manual total when no special filtering or trace is needed. `min([])` and `max([])` raise `ValueError` because an empty collection has no smallest or largest item. Supply a policy when empty input is expected:

```python
values = []
smallest = min(values) if values else None
assert smallest is None
```

Do not replace a rich loop with a pile of built-ins if that hides the rule. Use the simplest operation that expresses the entire required artifact.

Checkpoint: structured results

Structured result checkpoint

Choose the accumulator whose shape matches the promised artifact.

1. What does `counts[item] = counts.get(item, 0) + 1` do for a first occurrence?

- **A.** Uses zero, adds one, and stores count 1
- **B.** Deletes the item

- **C.** Stores zero without adding
- **D.** Raises KeyError before get can run

2. Why create `groups[key] = []` when a group key first appears?

- **A.** Dictionary values must always be lists
- **B.** That key needs its own collection before records can be appended
- **C.** It sorts the records
- **D.** It prevents future keys

3. Which result preserves first-seen unique order?

- **A.** A set alone
- **B.** A numeric total
- **C.** A list of first occurrences supported by a seen set
- **D.** A dictionary that overwrites every record

4. What happens when `min([])` is evaluated without a default policy?

- **A.** It returns 0
- **B.** It returns None
- **C.** It returns an empty list
- **D.** It raises ValueError because there is no smallest item

Write your choices before checking the answer key at the end of this book.

13. Do not change the list you are currently traversing

Removing items from a list while a `for` loop visits it can skip values because positions shift:

```python
numbers = [1, 2, 2, 3]

for number in numbers:
    if number == 2:
        numbers.remove(number)

print(numbers)
```

The result still contains a 2. After the first removal, the second 2 shifts into a position the loop has already advanced past.

Build a new result instead:

```python
numbers = [1, 2, 2, 3]
without_twos = []

for number in numbers:
    if number != 2:
        without_twos.append(number)

assert numbers == [1, 2, 2, 3]
assert without_twos == [1, 3]
```

Preserving the source also makes before-and-after assertions possible.

14. Build an expedition report

Start with this source evidence:

```python
observations = [
    {"station": "north", "species": "owl", "count": 2, "valid": True},
    {"station": "east", "species": "fox", "count": 1, "valid": True},
    {"station": "north", "species": "owl", "count": -1, "valid": False},
    {"station": "south", "species": "moth", "count": 4, "valid": True},
    {"station": "east", "species": "owl", "count": 3, "valid": True},
]

accepted = []
rejected = []
total_animals = 0
species_counts = {}
by_station = {}
unique_species = set()
```

In one `for` loop:

1. place valid non-negative records in `accepted` and all others in `rejected`;
2. update the total only for accepted records;
3. add each accepted record's count to `species_counts`;
4. group accepted species names by station, preserving observation order; and
5. collect unique accepted species.

Then derive `station_lines` with `enumerate(..., start=1)` over the station keys in their first-seen order. Run:

```python
assert observations[2]["count"] == -1
assert len(accepted) == 4
assert rejected == [observations[2]]
assert total_animals == 10
assert species_counts == {"owl": 5, "fox": 1, "moth": 4}
assert by_station == {
    "north": ["owl"],
    "east": ["fox", "owl"],
    "south": ["moth"],
}
assert unique_species == {"owl", "fox", "moth"}
assert station_lines == ["1. north", "2. east", "3. south"]
```

Hint: let the validity branch own every update

Append rejected records in the first branch and accepted records in the other. Only the accepted branch should update totals, counters, groups, and the set. Use `.get(species, 0)` for totals by species, and create a station list only when its key is new.

Show one complete solution after attempting the lab

```python
observations = [
    {"station": "north", "species": "owl", "count": 2, "valid": True},
```

PYTHON (CONTINUED)

```
    {"station": "east", "species": "fox", "count": 1, "valid": True},
    {"station": "north", "species": "owl", "count": -1, "valid": False},
    {"station": "south", "species": "moth", "count": 4, "valid": True},
    {"station": "east", "species": "owl", "count": 3, "valid": True},
]

accepted = []
rejected = []
total_animals = 0
species_counts = {}
by_station = {}
unique_species = set()

for observation in observations:
    if not observation["valid"] or observation["count"] < 0:
        rejected.append(observation)
    else:
        accepted.append(observation)
        total_animals += observation["count"]

        species = observation["species"]
        species_counts[species] = (
            species_counts.get(species, 0) + observation["count"]
        )
        unique_species.add(species)

        station = observation["station"]
        if station not in by_station:
            by_station[station] = []
        by_station[station].append(species)

station_lines = []
for number, station in enumerate(by_station, start=1):
    station_lines.append(f"{number}. {station}")
```

All result updates are controlled by the acceptance decision, and the original records remain available for comparison.

15. Explain the accumulator choices

1 What does each accumulator mean before the first item arrives?

2 Why does zero count as one accepted reading while adding zero to a total?

3 How does collecting rejected records improve the audit trail?

4 Why does each group key need its own list?

5 When is `sum()` clearer than a manual loop, and when does a loop reveal needed filtering or evidence?

Key points

Key points

- Initialize a result once before traversal with a value that correctly describes empty input.
- A total adds values; a count adds one per match; transformation and filtering produce different numbers of outputs per input.
- Lists preserve result order, dictionaries count or group by key, and sets preserve uniqueness.

- State what an accumulator means after each iteration; that invariant guides initialization and updates.
- Preserve rejected evidence when loss matters, and avoid mutating the list being traversed.
- Prefer a built-in when it clearly states the whole job and its empty behavior is acceptable. :::

References

- Python tutorial: `for` statements
- Python tutorial: looping techniques
- Python built-in functions: `enumerate`, `sum`, `min`, and `max`
- Python standard types: mapping operations

Repeating Until Something Changes

Online lesson and runnable notebook

Design while loops with explicit initial state, progress, termination, sentinel handling, iteration limits, and observable exit reasons.

- **Level:** Python Foundations · Unit 4
- **Estimated time:** 3.5–5 hours
- **You will learn:** Choose `while` for state-driven repetition, prove how every path progresses, handle sentinels and retry budgets, and report why the loop stopped.
- **Practice in:** Google Colab, JupyterLab, or a local editor

1. Use `while` when the number of repetitions is not supplied

A docking controller sends calibration pulses until its alignment error is small enough:

```python
error = 9
pulses = 0

while error > 2:
    error -= 2
    pulses += 1
    print(f"pulse {pulses}: error {error}")

assert error <= 2
assert pulses == 4
```

Unlike `for`, the loop is not consuming a prepared collection. It rechecks a condition whose state changes in the body.

Every dependable `while` loop answers four questions:

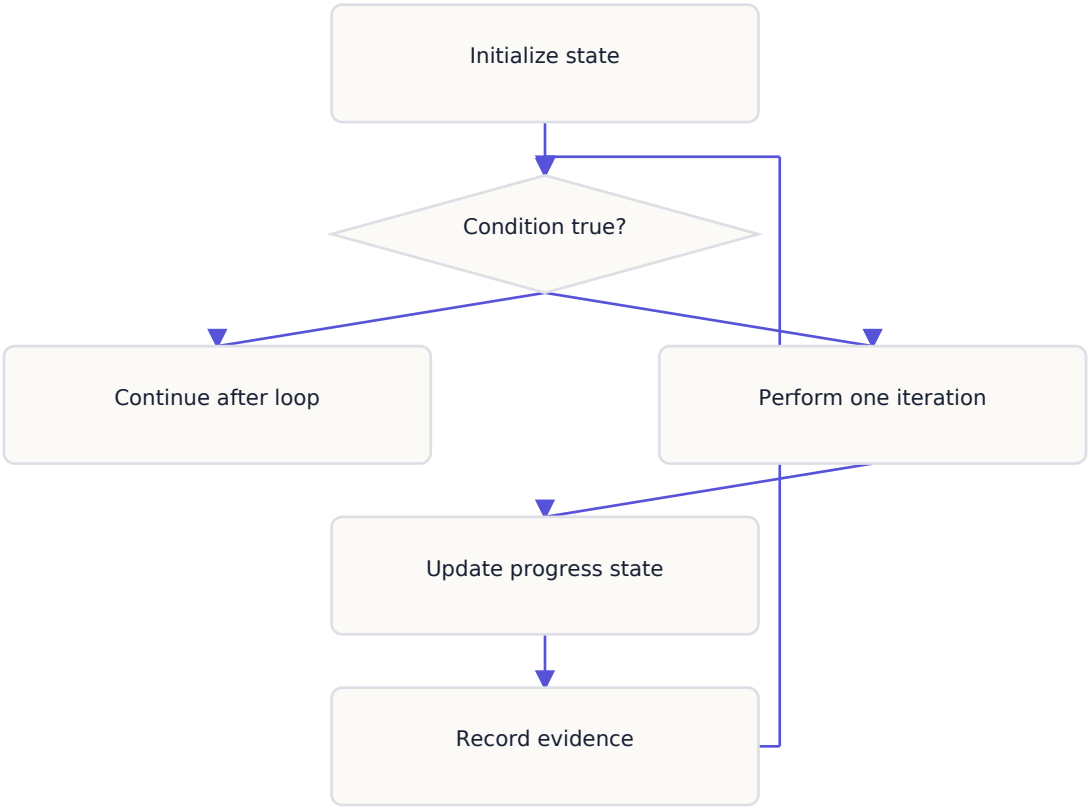
1. **Initial state:** What values exist before the first check?
2. **Continuation condition:** Exactly when may another iteration begin?
3. **Progress:** Which update moves the state toward a stopping condition?
4. **Termination:** Why must some future check become false or another explicit exit occur?

If one answer is missing, the loop deserves more design before execution.

Questions this lesson will answer

- How do zero iterations and boundary overshoot affect the result?
- How can an iteration cap turn “try until success” into a bounded process?
- What is a sentinel, and how does it differ from normal data?
- How can several stopping conditions produce one clear exit reason?
- Why is `continue` especially dangerous before a progress update?

A while loop checks before every iteration, updates state in its body, and must eventually reach a false check or an explicit exit.



Condition true? → Continue after loop: No

Condition true? → Perform one iteration: Yes

2. The condition is checked before the first iteration

A `while` loop may run zero times:

```python
error = 2
pulses = 0

while error > 2:
    error -= 2
    pulses += 1

assert pulses == 0
assert error == 2
```

The initial condition is false, so Python skips the body. Initialize any result needed later before the loop, just as with `for` accumulators.

Trace a countdown at its boundary:

```python
count = 3
visited = []
```

PYTHON (CONTINUED)

```
while count > 0:
    visited.append(count)
    count -= 1

assert visited == [3, 2, 1]
assert count == 0
```

The sequence of checks is true for 3, 2, and 1, then false for 0. The value 0 is not appended because the condition is checked first.

Check	`count > 0`	Body receives	New count
1	true	3	2
2	true	2	1
3	true	1	0
4	false	—	0

3. Progress may overshoot the exact target

A loop does not promise to land exactly on a boundary:

PYTHON

```
position = 0
step = 3

while position < 10:
    position += step

assert position == 12
assert position >= 10
```

The correct postcondition is `position >= 10`, not `position == 10`. Starting at 0 and adding 3 cannot reach 10 exactly. Choose the continuation condition and final assertion from the task:

- “continue while below 10” permits overshoot;
- “visit values no greater than 10” requires deciding whether to check the next position before applying it;
- “reach exactly 10” may require a different step rule.

Off-by-one bugs often come from asserting an exact value the progress operation cannot guarantee.

Checkpoint: checks and progress

While-loop progress checkpoint

Trace the check that happens before each possible iteration.

1. When does a `while` loop run zero times?

- **A.** When its condition is falsy at the first check
- **B.** Only when the body contains `pass`

- **C.** Whenever the state is numeric
- **D.** After exactly one update

2. For `count = 3` and `while count > 0`, which values enter the body if count decreases by one afterward?

- **A.** 3, 2, 1, 0
- **B.** 3, 2, 1
- **C.** 2, 1, 0
- **D.** 3 only

3. Starting at 0 and adding 3 while position is below 10 produces which final position?

- **A.** 9
- **B.** 10
- **C.** 12
- **D.** The loop cannot stop

4. Which question is essential for every while-loop path?

- **A.** Does the body print?
- **B.** Is the target a list?
- **C.** Can the variable be renamed?
- **D.** What moves this state toward stopping?

Write your choices before checking the answer key at the end of this book.

4. Recognize and interrupt a non-terminating loop

This broken shape never changes the value in its condition:

PYTHON

```
count = 3
iterations = 0

# The safety cap makes the demonstration terminate.
while count > 0 and iterations < 4:
    print("count is still", count)
    iterations += 1

assert count == 3
assert iterations == 4
```

Without the temporary `iterations < 4` guard, `count > 0` would remain true forever. In a notebook, use the stop/interrupt control rather than waiting. In a terminal, `Ctrl+C` commonly requests interruption.

After interrupting:

1. inspect every name used by the condition;
2. mark every path through the body;
3. find the update or explicit exit on each path; and

4 add a small trace or temporary cap before running again.

A permanent iteration limit can also be part of the real contract, not merely a debugging device.

5. Retry budgets bound an uncertain process

Suppose a docking system receives prepared measurements one at a time. It may succeed early or exhaust three attempts:

```python
measurements = [8, 5, 2, 1]
index = 0
attempts = 0
aligned = False
max_attempts = 3

while index < len(measurements) and attempts < max_attempts and not aligned:
    error = measurements[index]
    index += 1
    attempts += 1
    aligned = error <= 2

assert aligned is True
assert attempts == 3
assert index == 3
```

Three facts can stop the loop: no measurements remain, the attempt budget is exhausted, or alignment succeeds. Both `index` and `attempts` progress on every iteration. The fourth reading is not consumed after success.

Choose an exit reason afterward:

```python
if aligned:
    exit_reason = "aligned"
elif attempts >= max_attempts:
    exit_reason = "attempt limit"
else:
    exit_reason = "measurements exhausted"

assert exit_reason == "aligned"
```

The order matters if two facts become true on the final iteration. The most meaningful successful result is reported first.

6. Sentinels end a stream without becoming data

A **sentinel** is a special value that means “stop,” not an ordinary record. In a prepared sequence, an index lets us model a stream without introducing input or exceptions yet:

```python
commands = ["north", "scan", "STOP", "east"]
index = 0
processed = []

while index < len(commands):
    command = commands[index]
    index += 1

    if command == "STOP":
        exit_reason = "sentinel received"
```

PYTHON (CONTINUED)

```
        break

    processed.append(command)
else:
    exit_reason = "commands exhausted"

assert processed == ["north", "scan"]
assert exit_reason == "sentinel received"
assert index == 3
```

The index update occurs immediately after reading, so every iteration consumes one item. The sentinel is checked before appending, so it does not enter the data. `break` is developed fully in Lesson 6; here it makes the distinct stop event visible. The loop's `else` runs only if no `break` occurs.

Checkpoint: bounded stopping

Bounded stopping checkpoint

Identify every possible reason the loop can finish.

1. When is a maximum-attempt counter part of the real algorithm?

- **A.** When the process must stop after a bounded number of failures
- **B.** Only when printing a trace
- **C.** Whenever a for loop exists
- **D.** When no state changes

2. What does a sentinel value represent?

- **A.** The largest valid data item
- **B.** A special signal that ends processing rather than ordinary data
- **C.** An automatic exception
- **D.** The loop's initial value

3. Why record an exit reason after a loop with several stopping conditions?

- **A.** It forces another iteration
- **B.** It changes while into for
- **C.** It distinguishes success, exhaustion, limits, and other outcomes
- **D.** It guarantees every input is valid

4. Where should a sentinel check occur if the sentinel must not enter `processed`?

- **A.** After appending it
- **B.** Only before the loop starts
- **C.** After all commands are exhausted
- **D.** Before appending the current command

Write your choices before checking the answer key at the end of this book.

7. `continue` must not skip the progress update

This is a dangerous while-loop shape:

```python
values = [3, -1, 5]
index = 0
steps = 0

# A safety cap prevents the demonstration from hanging.
while index < len(values) and steps < 5:
    steps += 1
    value = values[index]
    if value < 0:
        continue
    index += 1
```

At index 1, the negative value triggers `continue` before `index += 1`. Every later iteration sees the same value. The safety cap ends the demonstration, but the intended traversal is broken.

Move unavoidable progress before any early continuation:

```python
values = [3, -1, 5]
index = 0
accepted = []

while index < len(values):
    value = values[index]
    index += 1

    if value < 0:
        continue

    accepted.append(value)

assert accepted == [3, 5]
assert index == len(values)
```

Or invert the condition and avoid `continue`:

```python
index = 0
accepted = []

while index < len(values):
    value = values[index]
    index += 1
    if value >= 0:
        accepted.append(value)
```

Use whichever shape makes progress easiest to verify.

8. Prefer `for` when a collection already determines the repetitions

The previous indexed loop can be simpler:

```python
accepted = []
```

PYTHON (CONTINUED)

```
for value in values:
    if value >= 0:
        accepted.append(value)
```

Choose `for` when the main job is “for every item supplied by this collection.” Choose `while` when the main job is “repeat while this changing state allows it,” such as retrying, advancing until a target, or consuming until a sentinel.

Do not use `while` merely because it can imitate a `for` loop. Manual index state adds an off-by-one and non-termination risk that direct traversal avoids.

9. Multiple progress variables need a shared trace

A simulation may update position, energy, and step count:

PYTHON

```
position = 0
energy = 7
steps = 0
history = []

while position < 5 and energy >= 2:
    before = (position, energy)
    position += 2
    energy -= 2
    steps += 1
    history.append((before, (position, energy)))

assert position == 6
assert energy == 1
assert steps == 3
```

The loop stops with both `position >= 5` and `energy < 2`. Decide which result has reporting precedence. The history proves each iteration changed the required state and makes overshoot visible.

Checkpoint: safe loop design

Safe while-loop design checkpoint

Choose the design that makes progress and stopping easiest to verify.

1. Where should an index update go when a later `continue` may run?

- **A.** Before any path that can continue
- **B.** Only after the loop
- **C.** Inside the false initial condition
- **D.** Nowhere; indexes update automatically

2. Which job normally favors a `for` loop over a manual indexed `while` loop?

- **A.** Retry until success or a limit
- **B.** Process every item already supplied by a list
- **C.** Advance until energy is empty
- **D.** Read until a sentinel

3. Position reaches its target on the same iteration that energy runs out. What should the program do?

- **A.** Pretend only one fact is true
- **B.** Run forever
- **C.** Apply a documented exit-reason precedence and preserve both final facts
- **D.** Delete the final state

4. Which trace best diagnoses a while loop?

- **A.** Only its final print
- **B.** Only the source line number
- **C.** A list of Python keywords
- **D.** Condition state and progress values before and after each iteration

Write your choices before checking the answer key at the end of this book.

10. Build a docking controller

Use a prepared measurement stream so the lab remains deterministic:

```python
measurements = [9, None, 6, 3, 2, 1]
max_attempts = 4
threshold = 2

index = 0
attempts = 0
skipped = 0
history = []
aligned = False
exit_reason = None
```

Implement this contract:

1. Inspect measurements in order while data remains, attempts remain, and the controller is not aligned.
2. Reading one list position always advances `index`.
3. `None` is a missing measurement: increment `skipped`, record `"missing measurement"`, and do not use an attempt.
4. A numeric measurement uses one attempt and adds a history string containing its attempt number and error.
5. Alignment succeeds when error is at most `threshold` and stops immediately.
6. After the loop, report one reason in this precedence: `"aligned"`, `"attempt limit"`, or `"measurements exhausted"`.
7. Every possible body path must make progress.

Run:

```python
assert index == 5
assert attempts == 4
assert skipped == 1
assert aligned is True
assert exit_reason == "aligned"
assert history == [
    "attempt 1: error 9",
    "missing measurement",
    "attempt 2: error 6",
    "attempt 3: error 3",
    "attempt 4: error 2",
]
```

Then test `[]`, `[None, None]`, `[7, 6, 5, 4, 1]` with three attempts, and a first measurement already within the threshold.

Hint: consume first, then classify the measurement

At the top of each iteration, copy `measurements[index]` into `measurement` and increment `index`. Handle `None` with a history append and `continue`; the index has already progressed. Numeric values increment attempts and may set aligned. Choose the reason only after the loop.

Show one complete solution after attempting the lab

```python
measurements = [9, None, 6, 3, 2, 1]
max_attempts = 4
threshold = 2

index = 0
attempts = 0
skipped = 0
history = []
aligned = False
exit_reason = None

while (
    index < len(measurements)
    and attempts < max_attempts
    and not aligned
):
    measurement = measurements[index]
    index += 1

    if measurement is None:
        skipped += 1
        history.append("missing measurement")
        continue

    attempts += 1
    history.append(f"attempt {attempts}: error {measurement}")
    if measurement <= threshold:
        aligned = True

if aligned:
    exit_reason = "aligned"
elif attempts >= max_attempts:
    exit_reason = "attempt limit"
else:
    exit_reason = "measurements exhausted"
```

The index progresses even on the `continue` path, while only numeric measurements consume the attempt budget.

11. Explain why it stops

1. What is the initial state and what could make the body run zero times?
2. Which names make progress on every path through the docking loop?
3. Why can a valid final position overshoot its target?
4. How does a sentinel differ from rejected ordinary data?
5. Why should an exit reason be chosen from final state rather than guessed from the last printed line?

Key points

Key points

- `while` is a pre-test loop for repetition controlled by changing state rather than a supplied collection.
- State the initial state, continuation condition, progress operation, and reason termination is guaranteed.
- Zero iterations and overshoot are normal possibilities that need explicit postconditions.
- Retry limits and sentinels provide bounded, meaningful stop conditions.
- Update essential progress before any `continue` path can skip it.
- Record final state and an exit reason when several conditions can stop the loop. :::

References

- Python language reference: the `while` statement
- Python tutorial: `break`, `continue`, and loop `else`
- Python built-in exception: `KeyboardInterrupt`

Searching, Skipping, and Stopping Loops

Online lesson and runnable notebook

Design searches for first, last, all, or existence results; use break, continue, and loop else deliberately; and preserve an explicit exit reason.

- **Level:** Python Foundations · Unit 4
- **Estimated time:** 3.5–5 hours
- **You will learn:** Define the search result before looping, skip unusable values, stop after sufficient evidence, and explain loop `else` and nearest-loop behavior.
- **Practice in:** Google Colab, JupyterLab, or a local editor

1. Define what “find” means before searching

A signal scanner receives several readings:

```python
signals = [
    {"code": "A1", "strength": 2},
    {"code": "B7", "strength": 9},
    {"code": "B7", "strength": 6},
]
```

“Find B7” is incomplete. The required artifact might be:

- the **first** matching record;
- the **last** matching record;
- **all** matching records;
- whether **any** match exists; or
- the position of the first match.

Each contract implies different loop behavior. A first-match search can stop as soon as it has sufficient evidence:

```python
first_match = None

for signal in signals:
    if signal["code"] == "B7":
        first_match = signal
        break

assert first_match == {"code": "B7", "strength": 9}
```

`break` exits the loop immediately. The final record is never inspected, which is correct only because later matches cannot change the first result.

Questions this lesson will answer

- When should a search stop, and when must it inspect every item?
- How does `continue` differ from `break` and `pass`?
- What does a loop's `else` really mean?
- Which loop receives a `break` inside nested code?
- When should membership or `any()` replace a manual search?

2. First, last, and all matches require different updates

A last-match search must keep traversing:

```python
last_match = None

for signal in signals:
    if signal["code"] == "B7":
        last_match = signal

assert last_match == {"code": "B7", "strength": 6}
```

Every new match replaces the prior one. An all-match search collects instead:

```python
all_matches = []

for signal in signals:
    if signal["code"] == "B7":
        all_matches.append(signal)

assert all_matches == [signals[1], signals[2]]
```

An existence search needs only a Boolean:

```python
found = False

for signal in signals:
    if signal["code"] == "B7":
        found = True
        break

assert found is True
```

State the no-match result too: `None` for no selected record, `[]` for no collected records, and `False` for no existence evidence. Empty input should produce the same defined values without special repair after the loop.

3. `continue` rejects one iteration, not the whole search

Suppose malformed records lack a usable code:

```python
signals = [
    {"code": ""},
    {"strength": 8},
    {"code": "B7", "strength": 9},
]
```

PYTHON (CONTINUED)

```
rejected = []
first_match = None

for position, signal in enumerate(signals):
    if "code" not in signal or not signal["code"]:
        rejected.append((position, signal))
        continue

    if signal["code"] == "B7":
        first_match = signal
        break

assert len(rejected) == 2
assert first_match == signals[2]
```

`continue` skips the rest of the current body and asks the `for` loop for its next item. Recording the rejection before continuing prevents silent loss.

`break` would be wrong for a bad first record because later valid records still need inspection. Conversely, `continue` would be wrong after a first-match success because it would keep searching after the contract is satisfied.

Checkpoint: search contracts

Search contract checkpoint

Choose the loop behavior from the exact result required.

1. Why can a first-match search break after assigning its result?

- **A.** Later items cannot change which match was first
- **B.** break sorts the remaining items
- **C.** Every search must stop after one item
- **D.** The result is always the final item

2. How does a last-match search normally behave?

- **A.** Break at the first match
- **B.** Keep traversing and replace the selected result at each match
- **C.** Append only rejected records
- **D.** Stop before reading input

3. What does `continue` do after recording one unusable record in a for loop?

- **A.** Ends the entire program
- **B.** Restarts the list from the beginning
- **C.** Skips the remaining body for that record and requests the next item
- **D.** Deletes the record from the source

4. What is a natural no-match result for an all-matches search?

- **A.** The first source item
- **B.** True

- **C.** An undefined name
- **D.** An empty list

Write your choices before checking the answer key at the end of this book.

4. Loop `else` means no `break` occurred

A loop may have an `else` aligned with it:

```
PYTHON
codes = ["A1", "C3", "B7"]
target = "B7"

for code in codes:
    if code == target:
        result = f"found {target}"
        break
else:
    result = f"{target} not found"

assert result == "found B7"
```

The `else` runs only if the loop finishes **without** `break`. It does not mean the final `if` condition was false. Several nonmatching iterations may occur, but a later break still skips the loop `else`.

For no match:

```
PYTHON
codes = ["A1", "C3"]
target = "B7"

for code in codes:
    if code == target:
        result = f"found {target}"
        break
else:
    result = f"{target} not found"

assert result == "B7 not found"
```

An empty collection also reaches `else` because no `break` occurred.

Attach the meaning to `break`

Read loop `else` as “normal exhaustion.” Use it when `break` has one clear meaning such as success. If several unrelated breaks exist, an explicit exit-reason name may be easier to understand. :::

5. A `while` loop can also have `else`

```python
values = [3, 5, 8]
index = 0

while index < len(values):
    if values[index] % 2 == 0:
        first_even = values[index]
        break
    index += 1
else:
    first_even = None

assert first_even == 8
```

The `else` runs if the condition becomes false normally. A `break` skips it. As in Lesson 5, every path that stays in the loop must advance `index`; here the break path exits, and the nonmatch path increments.

6. Record why a loop exited

One scanner can encounter success, shutdown, exhaustion, or an invalid limit:

```python
commands = ["noise", "target", "shutdown"]
exit_reason = "exhausted"
found = None

for command in commands:
    if command == "shutdown":
        exit_reason = "shutdown"
        break
    if command == "target":
        found = command
        exit_reason = "target found"
        break

assert exit_reason == "target found"
assert found == "target"
```

Initialize the normal-exhaustion reason before the loop; replace it immediately before each break. This leaves an explicit result after all paths.

A loop `else` would also work for exhaustion, but it would not distinguish two kinds of break by itself. Choose the form that makes exit states most visible.

Checkpoint: loop `else` and exit reasons

Loop exit checkpoint

Trace whether a break occurred and which result remains afterward.

1. When does the `else` attached to a `for` loop run?

- **A.** When traversal finishes without a break
- **B.** After every continue
- **C.** Only after the first match
- **D.** Whenever the collection is non-empty

2. What happens to loop `else` when a for loop receives an empty list?

- **A.** Python raises a loop error
- **B.** It runs because no break occurred
- **C.** It is skipped because no body ran
- **D.** It creates one placeholder item

3. Why can an explicit `exit_reason` help when a loop has breaks for success and shutdown?

- **A.** It makes break exit two loops
- **B.** It disables else
- **C.** It preserves which event caused the exit
- **D.** It repeats the final iteration

4. When is a while loop's `else` skipped?

- **A.** When the condition eventually becomes false
- **B.** When it runs zero times
- **C.** After every normal iteration
- **D.** When a `break` exits that loop

Write your choices before checking the answer key at the end of this book.

7. `break` exits only the nearest loop

Nested loops make the target important:

```python
groups = [
    ["noise", "target", "noise"],
    ["target", "noise"],
]
visited = []

for group_number, group in enumerate(groups):
    for signal in group:
        visited.append((group_number, signal))
        if signal == "target":
            break

assert visited == [
    (0, "noise"),
    (0, "target"),
    (1, "target"),
]
```

Each `break` exits only the inner loop. The outer loop proceeds to the next group. If the whole search should stop, preserve a flag and check it in the outer loop:

```python
found = None

for group_number, group in enumerate(groups):
    for signal in group:
        if signal == "target":
```

PYTHON (CONTINUED)

```
            found = (group_number, signal)
            break
    if found is not None:
        break

assert found == (0, "target")
```

Lesson 7 applies this behavior to grids.

8. `pass` does nothing; it does not skip the body

`pass` is a placeholder statement:

PYTHON

```
for value in [1, 2, 3]:
    if value == 2:
        pass
    print(value)
```

All three values print. For value 2, `pass` performs no action, then execution continues with the next statement in the same body.

With `continue`, the value 2 would not print:

PYTHON

```
for value in [1, 2, 3]:
    if value == 2:
        continue
    print(value)
```

Use `pass` only when Python requires a statement but the block is intentionally empty during development or by design. It is not a synonym for “skip this item.”

9. Prefer direct operations for direct questions

Manual loops are excellent for learning and for collecting rich evidence. When the final question is already a built-in operation, say it directly:

PYTHON

```
codes = ["A1", "C3", "B7"]

assert "B7" in codes
assert any(code.startswith("C") for code in codes)
assert all(len(code) == 2 for code in codes)
```

- Membership answers whether an equal value exists.
- `any(...)` answers whether at least one generated fact is truthy.
- `all(...)` answers whether every generated fact is truthy.

The expressions inside `any` and `all` are **generator expressions**, a lazy form previewed here because the built-ins consume it directly. Lesson 8 focuses on comprehensions and explains why square brackets would unnecessarily build an intermediate list for this yes/no question.

Use an explicit loop when you need the matching record, position, rejection log, or exit reason—not only a Boolean.

Checkpoint: control statements and direct questions

Control choice checkpoint

Choose the statement or operation whose behavior matches the task.

1. What does `break` inside the inner of two nested loops exit?

- **A.** Only the nearest inner loop
- **B.** Both loops automatically
- **C.** The complete Python process
- **D.** Neither loop

2. What does `pass` do inside an if body?

- **A.** Skips the rest of the loop body
- **B.** Nothing; execution continues with the following statement
- **C.** Exits the nearest loop
- **D.** Repeats the current item

3. You need only to know whether exact string `B7` exists in a list. What is the clearest operation?

- **A.** A while loop with two counters
- **B.** A nested loop
- **C.** `'B7' in codes`
- **D.** Collect every nonmatch

4. When is an explicit search loop more useful than membership alone?

- **A.** When no input exists
- **B.** When every value is the same
- **C.** When syntax must be longer
- **D.** When you need the matching record, its position, or rejection evidence

Write your choices before checking the answer key at the end of this book.

10. Build a signal scanner

Use this ordered signal stream:

PYTHON

```
signals = [
    {"code": "", "strength": 8},
    {"strength": 9},
    {"code": "A1", "strength": 4},
    {"code": "B7", "strength": 3},
    {"code": "B7", "strength": 9},
    {"code": "SHUTDOWN", "strength": 0},
    {"code": "B7", "strength": 10},
]
target = "B7"
```

PYTHON (CONTINUED)

```
minimum_strength = 7

rejected = []
first_match = None
first_position = None
exit_reason = "exhausted"
```

Implement this contract:

1. Reject a record when the code key is absent or the code is empty; preserve its position and reason, then continue.
2. Stop with reason `"shutdown"` when code is `"SHUTDOWN"`.
3. A target qualifies only when strength is at least the minimum.
4. Stop at the first qualifying target, preserving the record and its position.
5. If traversal exhausts normally, keep reason `"exhausted"`.
6. Preserve the source list unchanged.

Run:

PYTHON

```
assert rejected == [
    (0, "empty code"),
    (1, "missing code"),
]
assert first_match == {"code": "B7", "strength": 9}
assert first_position == 4
assert exit_reason == "target found"
assert signals[-1]["strength"] == 10
```

Then test no target, shutdown before a target, an empty list, and a qualifying target in the first position. Explain whether loop `else` or the initialized exit reason better communicates your implementation.

Hint: order rejection, shutdown, and success checks

Use `enumerate(signals)` and make the missing-key check before indexing `code`. Give missing and empty codes distinct rejection tuples. Set `exit_reason` immediately before each `break`. A loop `else` can explicitly restore or confirm `"exhausted"`.

Show one complete solution after attempting the lab

PYTHON

```
signals = [
    {"code": "", "strength": 8},
    {"strength": 9},
    {"code": "A1", "strength": 4},
    {"code": "B7", "strength": 3},
    {"code": "B7", "strength": 9},
    {"code": "SHUTDOWN", "strength": 0},
    {"code": "B7", "strength": 10},
]
target = "B7"
minimum_strength = 7

rejected = []
first_match = None
first_position = None
```

PYTHON (CONTINUED)

```
exit_reason = "exhausted"

for position, signal in enumerate(signals):
    if "code" not in signal:
        rejected.append((position, "missing code"))
        continue
    if not signal["code"]:
        rejected.append((position, "empty code"))
        continue
    if signal["code"] == "SHUTDOWN":
        exit_reason = "shutdown"
        break
    if signal["code"] == target and signal["strength"] >= minimum_strength:
        first_match = signal
        first_position = position
        exit_reason = "target found"
        break
else:
    exit_reason = "exhausted"
```

The two target records at positions 3 and 4 demonstrate that matching the code is not enough; the strength rule still decides qualification.

11. Explain the stopping decision

1 Why can a first-match loop stop but an all-match loop cannot?

2 What evidence should be recorded before `continue` discards the current path?

3 Why does loop `else` mean no break rather than “the final if was false”?

4 How would you stop both loops in a nested search?

5 When do membership, `any()`, or `all()` express the entire question?

Key points

Key points

- Define whether the result is first, last, all, existence, or position before choosing loop control.
- `break` exits the nearest loop; `continue` skips the remaining current body; `pass` does nothing.
- A loop’s `else` runs after normal exhaustion and is skipped by `break`.
- Record rejection evidence before continuing and an exit reason before breaking when several exit states matter.
- Use membership, `any()`, or `all()` when a direct Boolean is the whole result; keep an explicit loop for richer artifacts. :::

References

- Python tutorial: `break`, `continue`, and loop `else`
- Python language reference: the `for` statement
- Python built-in function: `any`
- Python built-in function: `all`

Working Through Grids and Pairs

Online lesson and runnable notebook

Traverse rectangular and ragged grids, generate intended pairs, trace nested targets, control inner-loop exits, and recognize multiplied work.

- **Level:** Python Foundations · Unit 4
- **Estimated time:** 3.5–5 hours
- **You will learn:** Trace nested loops over grids and pairs, handle ragged rows, exit the intended loop, and estimate how often the inner body runs.
- **Practice in:** Google Colab, JupyterLab, or a local editor

1. A grid has an outer route and an inner route

A treasure map can be represented as a list of row strings:

```python
grid = [
    ".T.",
    "...",
    "T..",
]

treasures = []

for row_index, row in enumerate(grid):
    for column_index, cell in enumerate(row):
        if cell == "T":
            treasures.append((row_index, column_index))

assert treasures == [(0, 1), (2, 0)]
```

The outer loop receives one row. For that row, the inner loop visits every cell. Only after the inner loop is exhausted does the outer loop advance to the next row.

At each inner iteration, four values have a current meaning:

- `row_index`: which row the outer loop selected;
- `row`: the current row value;
- `column_index`: which position the inner loop selected; and
- `cell`: the character at that coordinate.

Coordinates use zero-based `(row, column)` order. State that convention because some domains write x/y or column/row instead.

Questions this lesson will answer

- How do you trace which target changes in the outer and inner loops?
- Why does `len(grid[0])` fail as a rule for ragged data?
- How do Cartesian pairs differ from aligned pairs made with `zip()`?

- What does an inner `break` leave running?
- How can work accidentally multiply inside nested loops?

2. Trace one complete outer iteration at a time

For a two-row grid:

```python
grid = ["AB", "CD"]
trace = []

for row_index, row in enumerate(grid):
    for column_index, cell in enumerate(row):
        trace.append((row_index, column_index, cell))

assert trace == [
    (0, 0, "A"),
    (0, 1, "B"),
    (1, 0, "C"),
    (1, 1, "D"),
]
```

The row index remains 0 while both cells in `"AB"` are visited. Then it becomes 1 and the column index begins again at 0 for `"CD"`.

Inner trip	Row index	Row	Column index	Cell
1	0	`"AB"`	0	`"A"`
2	0	`"AB"`	1	`"B"`
3	1	`"CD"`	0	`"C"`
4	1	`"CD"`	1	`"D"`

When nested output surprises you, write the outer value once and indent its inner trace rows underneath. Do not try to hold all target changes in memory.

3. Let each row supply its own columns

A **rectangular** grid has equal row lengths. A **ragged** grid does not:

```python
ragged = ["ABC", "D", "EF"]
coordinates = []

for row_index, row in enumerate(ragged):
    for column_index, cell in enumerate(row):
        coordinates.append((row_index, column_index, cell))

assert len(coordinates) == 6
assert coordinates[-1] == (2, 1, "F")
```

Directly traversing each `row` is safe for both shapes. This indexed version is not:

```python
ragged = ["ABC", "D", "EF"]
```

PYTHON (CONTINUED)

```
for row_index in range(len(ragged)):
    for column_index in range(len(ragged[0])):
        print(ragged[row_index][column_index])
```

The first row has three columns, so the inner range always includes indexes 0, 1, and 2. The second row has only index 0, and index 1 raises `IndexError`.

If a rectangular shape is required by the application, validate it explicitly:

PYTHON

```
grid = ["ABC", "DEF", "GHI"]
expected_width = len(grid[0]) if grid else 0
is_rectangular = all(len(row) == expected_width for row in grid)
assert is_rectangular is True
```

An empty grid receives width zero by policy and passes the “all rows match” fact. Decide separately whether the application permits an empty map.

Checkpoint: tracing grids

Grid traversal checkpoint

Trace which collection supplies each target.

1. In a nested grid traversal, what does the outer loop normally supply?

- **A.** One row at a time
- **B.** Every coordinate at once
- **C.** Only the final cell
- **D.** A sorted copy of the grid

2. What happens to the column index when the outer loop advances to a new row?

- **A.** It continues beyond the old row length
- **B.** The new inner enumerate starts again at zero
- **C.** It becomes the row string
- **D.** It is deleted before traversal

3. Why is `for cell in row` safe for ragged rows?

- **A.** Python pads every row
- **B.** Ragged rows cannot be strings
- **C.** Each row supplies only its own existing cells
- **D.** The first row controls all lengths

4. Why can using `range(len(grid[0]))` for every row fail?

- **A.** range works only once
- **B.** Strings have no length
- **C.** The outer loop skips row zero

- **D.** A shorter later row may not have all first-row positions

Write your choices before checking the answer key at the end of this book.

4. Flatten a nested collection with an explicit loop

Flattening turns a list of lists into one ordered list:

```python
packs = [["map", "rope"], [], ["torch"]]
all_items = []

for pack in packs:
    for item in pack:
        all_items.append(item)

assert all_items == ["map", "rope", "torch"]
```

The empty inner list contributes no iterations and needs no special case. The result preserves outer order, then inner order.

Do not call `.append(pack)` if the contract needs individual items; that would produce another nested list. `.extend(pack)` can flatten one level without an inner loop, but the explicit version is useful when each item needs filtering, transformation, or a coordinate.

5. Nested loops create Cartesian pairs

Two independent loops combine every left value with every right value:

```python
characters = ["mage", "scout"]
items = ["map", "key", "torch"]
loadout_pairs = []

for character in characters:
    for item in items:
        loadout_pairs.append((character, item))

assert len(loadout_pairs) == 6
```

This is a Cartesian product: 2 characters × 3 items = 6 pairs. `zip()` means a different relationship:

```python
assigned = list(zip(characters, items))
assert assigned == [("mage", "map"), ("scout", "key")]
```

`zip()` aligns first with first and second with second, then stops at the shorter input. Use nested loops for every combination; use `zip()` for already aligned positions.

6. Avoid self-pairs and mirrored duplicates

A tournament needs each distinct pair once. This naive loop includes self-pairs and both `(A, B)` and `(B, A)`:

```python
players = ["Ada", "Lin", "Mina"]
pairs = []
```

PYTHON (CONTINUED)

```
for left in players:
    for right in players:
        pairs.append((left, right))

assert len(pairs) == 9
```

Use positions so the right player always appears later:

PYTHON

```
pairs = []

for left_index, left in enumerate(players):
    for right in players[left_index + 1:]:
        pairs.append((left, right))

assert pairs == [
    ("Ada", "Lin"),
    ("Ada", "Mina"),
    ("Lin", "Mina"),
]
```

The slice excludes the current and earlier positions. For three players, the result has three unique unordered pairs.

Checkpoint: pairing relationships

Pair generation checkpoint

Choose whether the task requires every combination or aligned positions.

1. How many inner-body executions occur for 2 outer values and 3 inner values?

- **A.** 6
- **B.** 5
- **C.** 3
- **D.** 2

2. What relationship does `zip(names, scores)` represent?

- **A.** Every name with every score
- **B.** Values aligned by position, stopping at the shorter input
- **C.** Only duplicate values
- **D.** Pairs in reverse order automatically

3. Why use `players[left_index + 1:]` for tournament pairs?

- **A.** It sorts player names
- **B.** It repeats self-pairs
- **C.** It excludes the current and earlier players, avoiding self and mirrored pairs
- **D.** It changes the source list

4. What order does a standard two-loop flatten preserve?

- **A.** Alphabetical order

- **B.** Reverse inner order
- **C.** No predictable order
- **D.** Outer collection order, then each inner collection's order

Write your choices before checking the answer key at the end of this book.

7. An inner `break` leaves the outer loop active

Find the first treasure in each row:

```
PYTHON
grid = ["..T.T", ".....", ".T..."]
first_in_each_row = []

for row_index, row in enumerate(grid):
    for column_index, cell in enumerate(row):
        if cell == "T":
            first_in_each_row.append((row_index, column_index))
            break

assert first_in_each_row == [(0, 2), (2, 1)]
```

The first `break` stops scanning row 0 after column 2, but the outer loop still visits rows 1 and 2. A row with no treasure simply completes its inner loop.

To stop the entire search after the first treasure anywhere, use a flag or result checked by the outer loop:

```
PYTHON
first_treasure = None

for row_index, row in enumerate(grid):
    for column_index, cell in enumerate(row):
        if cell == "T":
            first_treasure = (row_index, column_index)
            break
    if first_treasure is not None:
        break

assert first_treasure == (0, 2)
```

The result name doubles as the signal. It starts `None` and becomes a coordinate only on success.

8. Inner loop `else` can report no row match

```
PYTHON
grid = ["..T", "...", ".T."]
row_results = []

for row_index, row in enumerate(grid):
    for column_index, cell in enumerate(row):
        if cell == "T":
            row_results.append((row_index, column_index))
            break
    else:
        row_results.append((row_index, None))

assert row_results == [(0, 2), (1, None), (2, 1)]
```

The `else` belongs to the inner `for`: it runs when that row exhausts without a break. Its indentation aligns with the inner loop, not the outer one.

9. Count how often the inner body runs

Nested loops multiply work. A 100-row by 100-column grid has 10,000 cell visits. That can be necessary; a full grid scan genuinely needs every cell. Problems arise when invariant work is repeated inside the inner body:

```python
rows = ["ABC", "DEF"]
allowed = ["A", "B", "C", "D", "E", "F"]
allowed_set = set(allowed)

matches = []
for row in rows:
    for cell in row:
        if cell in allowed_set:
            matches.append(cell)
```

Build `allowed_set` once before the loops. Rebuilding it for every cell would produce the same answer but repeat work that does not depend on the current row or cell.

Unit 7 will analyze algorithmic complexity formally. Here, ask a practical question: “How many times will this line run, and does it depend on either loop target?” Move loop-invariant preparation outward.

Checkpoint: exits and multiplied work

Nested exits and work checkpoint

Track which loop owns each control statement and how often work repeats.

1. After `break` runs inside an inner grid loop, what normally happens next?

- **A.** The outer loop may continue with its next row
- **B.** Both loops always stop
- **C.** The current row restarts
- **D.** Python deletes the grid

2. How can a nested search stop both loops after one result?

- **A.** Use continue twice
- **B.** Break the inner loop, then test the result and break the outer loop
- **C.** Put pass after the inner break
- **D.** Remove indentation

3. What does an `else` attached to the inner loop mean for one row?

- **A.** The outer loop ended
- **B.** The row was empty only
- **C.** The inner row traversal finished without break
- **D.** Every cell matched

4. Where should a set conversion go if it does not depend on either loop target?

- **A.** In the innermost body
- **B.** After every match
- **C.** Inside only the final row
- **D.** Before both loops

Write your choices before checking the answer key at the end of this book.

10. Survey a treasure grid

Use this ragged map:

```python
grid = [
    ".T..",
    "..#T.",
    "T",
    "....",
]

all_treasures = []
first_by_row = []
blocked_cells = []
cell_visits = 0
```

Implement this contract in one full traversal:

1. Count every visited cell.
2. Collect every treasure coordinate in row-major order.
3. Collect blocked `#` coordinates.
4. Record only the first treasure coordinate for each row that has one. Do not break the full traversal, because all treasures and blocks are still needed.
5. Afterward, create every unique pair of treasure coordinates without self or mirrored pairs.
6. Preserve the ragged source unchanged.

Run:

```python
assert cell_visits == 14
assert all_treasures == [(0, 1), (1, 3), (2, 0)]
assert first_by_row == [(0, 1), (1, 3), (2, 0)]
assert blocked_cells == [(1, 2)]
assert treasure_pairs == [
    ((0, 1), (1, 3)),
    ((0, 1), (2, 0)),
    ((1, 3), (2, 0)),
]
assert grid[2] == "T"
```

Then add a second treasure to row 0 and prove that `all_treasures` changes while `first_by_row` still has only one coordinate for that row.

Hint: use a per-row flag rather than breaking

At the beginning of each outer iteration, set `row_has_treasure = False`. When a cell is treasure, always append to `all_treasures`, but append to `first_by_row` only if the flag is still false; then set it true. Build unique pairs afterward using a slice that begins after the left position.

Show one complete solution after attempting the lab

PYTHON

```
grid = [
    ".T..",
    "..#T.",
    "T",
    "....",
]

all_treasures = []
first_by_row = []
blocked_cells = []
cell_visits = 0

for row_index, row in enumerate(grid):
    row_has_treasure = False
    for column_index, cell in enumerate(row):
        cell_visits += 1
        coordinate = (row_index, column_index)
        if cell == "T":
            all_treasures.append(coordinate)
            if not row_has_treasure:
                first_by_row.append(coordinate)
                row_has_treasure = True
        elif cell == "#":
            blocked_cells.append(coordinate)

treasure_pairs = []
for left_index, left in enumerate(all_treasures):
    for right in all_treasures[left_index + 1:]:
        treasure_pairs.append((left, right))
```

The per-row flag protects the “first” result without ending the traversal that other artifacts require.

11. Explain the nested paths

1 Which values remain constant during one complete inner traversal?

2 Why is direct row traversal safe for ragged shapes?

3 How do nested loops and `zip()` represent different pairing relationships?

4 Why would an inner break lose the later treasure evidence required by the lab?

5 Which work can be moved before both loops because it does not depend on either target?

Key points

Key points

- The outer loop selects a collection such as a row; the inner loop consumes that current collection before the outer target advances.
- Directly enumerate each row to support rectangular and ragged shapes safely.
- Nested independent loops form every combination; `zip()` aligns positions.

- `break` exits only the nearest loop. Use a result or flag when the outer loop must also stop or when traversal must continue for other artifacts.
- Nested work multiplies; prepare values outside any loop whose targets they do not depend on. :::

References

- Python tutorial: looping techniques
- Python built-in function: `enumerate`
- Python built-in function: `zip`
- Python tutorial: `break` and loop `else`

Building Collections with Comprehensions

Online lesson and runnable notebook

Translate explicit loops into readable list, set, and dictionary comprehensions while preserving evaluation order, collision behavior, scope, and readability.

- **Level:** Python Foundations · Unit 4
- **Estimated time:** 3–4.5 hours
- **You will learn:** Read and write list, set, and dictionary comprehensions, distinguish filters from conditional expressions, and choose an explicit loop when the compact form hides important work.
- **Practice in:** Google Colab, JupyterLab, or a local editor

1. A comprehension describes one new collection

An explicit loop can transform every reading:

```python
celsius_values = [0, 10, 25]
fahrenheit_values = []

for celsius in celsius_values:
    fahrenheit_values.append(celsius * 9 / 5 + 32)

assert fahrenheit_values == [32.0, 50.0, 77.0]
```

A list comprehension expresses the same result:

```python
fahrenheit_values = [
    celsius * 9 / 5 + 32
    for celsius in celsius_values
]
```

Read it in this order:

1. `for celsius in celsius_values` supplies each source value;
2. `celsius * 9 / 5 + 32` builds one output; and
3. the surrounding brackets collect those outputs into a new list.

The output expression appears first in the syntax, but the `for` clause tells you where its name comes from. A comprehension is useful when the complete result can be said as one short sentence: “the converted temperature for each source temperature.”

Questions this lesson will answer

- How does a trailing `if` filter inputs?
- How is a conditional expression different from that filter?
- What do set and dictionary comprehensions collect?

- What happens when several inputs produce the same dictionary key?
- When does a comprehension become less readable than the loop it replaces?

2. Translate back to a loop whenever the order is unclear

For this comprehension:

```python
names = ["  ADA ", "", "  Lin  "]
clean_names = [name.strip().title() for name in names]
```

The equivalent loop is:

```python
clean_names = []
for name in names:
    cleaned = name.strip().title()
    clean_names.append(cleaned)
```

Both produce `['Ada', '', 'Lin']`. Write the expanded form when debugging:

- the source clause becomes the loop header;
- the leading expression becomes the value appended; and
- a trailing filter, if present, becomes an `if` around the append.

The comprehension builds a new outer list. It does not mutate `names`.

3. A trailing `if` filters source items

```python
readings = [8, -2, 0, 5]
usable = [reading for reading in readings if reading >= 0]

assert usable == [8, 0, 5]
assert readings == [8, -2, 0, 5]
```

Read the execution order as:

1. receive a `reading`;
2. check `reading >= 0`;
3. only when true, evaluate and collect the leading `reading` expression.

Transformation and filtering can coexist:

```python
scaled = [reading * 10 for reading in readings if reading >= 0]
assert scaled == [80, 0, 50]
```

The condition protects the output expression. For example, filter nonzero values before division:

```python
values = [4, 0, 2]
reciprocals = [1 / value for value in values if value != 0]
assert reciprocals == [0.25, 0.5]
```

No division occurs for zero.

Checkpoint: reading list comprehensions

List comprehension checkpoint

Translate the compact expression into its loop behavior.

1. In `[value * 2 for value in values]`, what is collected for each source item?

- **A.** `value * 2`
- **B.** The name `values`
- **C.** Only Boolean results
- **D.** Nothing until break

2. What does the trailing clause do in `[x for x in values if x >= 0]`?

- **A.** Chooses between two output values
- **B.** Excludes source items whose condition is false
- **C.** Stops after the first nonnegative value
- **D.** Mutates the source list

3. Why is `[1 / x for x in values if x != 0]` safe for zero inputs?

- **A.** Division by zero returns None in comprehensions
- **B.** The list removes errors afterward
- **C.** The filter rejects zero before evaluating `1 / x` for that item
- **D.** The output is always an integer

4. What normally happens to the source list when a comprehension builds transformed immutable values?

- **A.** It becomes empty
- **B.** It receives the outputs too
- **C.** It is sorted
- **D.** It remains unchanged

Write your choices before checking the answer key at the end of this book.

4. A conditional expression chooses an output for every item

This form has an `if` and `else` before the `for`:

PYTHON
```
scores = [82, 40, 71]
labels = ["pass" if score >= 70 else "retry" for score in scores]

assert labels == ["pass", "retry", "pass"]
assert len(labels) == len(scores)
```

Every score produces exactly one label. Compare it with a trailing filter:

```python
passing_scores = [score for score in scores if score >= 70]
assert passing_scores == [82, 71]
```

- `result_when_true if condition else result_when_false` is a **conditional expression**. It transforms every input into one of two outputs.
- `... for item in source if condition` is a **filter**. It may omit the input.

The location and presence of `else` reveal the difference.

A conditional expression and filter can both appear:

```python
values = [-3, 0, 4, None]
labels = [
    "positive" if value > 0 else "not positive"
    for value in values
    if value is not None
]

assert labels == ["not positive", "not positive", "positive"]
```

First the source supplies a value. The trailing filter excludes `None`. Then the conditional expression chooses a label for each remaining value.

5. Set comprehensions collect unique results

```python
species = [" Owl ", "fox", "owl", "FOX"]
normalized = {name.strip().lower() for name in species}

assert normalized == {"owl", "fox"}
```

Curly braces with one expression create a set comprehension. Multiple inputs can produce the same normalized value, and the set keeps it once. Do not rely on a set's display order; use a list when first-seen order matters.

An empty set comprehension still returns a set:

```python
none_long = {name for name in species if len(name.strip()) > 10}
assert none_long == set()
```

The literal `{}` is an empty dictionary, so `set()` remains the syntax for a standalone empty set.

6. Dictionary comprehensions require a key and value

```python
stations = [
    {"name": "north", "reading": 4},
    {"name": "east", "reading": 7},
]
by_name = {
    station["name"]: station["reading"]
    for station in stations
}

assert by_name == {"north": 4, "east": 7}
```

The expression before the `for` has `key: value`. The equivalent loop assigns one dictionary entry per source record:

```python
by_name = {}
for station in stations:
    by_name[station["name"]] = station["reading"]
```

A dictionary comprehension can transform both sides:

```python
raw = {" North ": 4, " EAST ": 7}
clean = {name.strip().lower(): reading * 10 for name, reading in raw.items()}
assert clean == {"north": 40, "east": 70}
```

Checkpoint: output collection shapes

Comprehension shape checkpoint

Choose the output cardinality and collection behavior.

1. How many outputs does `['yes' if x else 'no' for x in values]` produce?

- **A.** One for every source item
- **B.** Only one total
- **C.** Only for truthy items
- **D.** None for falsy items

2. What happens when several source strings normalize to the same value in a set comprehension?

- **A.** Python raises a duplicate error
- **B.** The result keeps one equal value
- **C.** The source strings are deleted
- **D.** The result becomes a list

3. Which expression identifies a dictionary comprehension?

- **A.** One expression inside parentheses
- **B.** One expression inside braces

- **C.** A `key: value` expression before the for clause
- **D.** A trailing else with no condition

4. What does a set comprehension with no accepted items return?

- **A.** `{}` as an empty dictionary
- **B.** None
- **C.** An empty tuple
- **D.** An empty set equal to `set()`

Write your choices before checking the answer key at the end of this book.

7. Duplicate dictionary keys keep the last assigned value

```python
records = [
    {"station": "north", "reading": 3},
    {"station": "east", "reading": 7},
    {"station": "north", "reading": 9},
]
latest = {record["station"]: record["reading"] for record in records}

assert latest == {"north": 9, "east": 7}
```

The second `"north"` assignment replaces the first. That may be a deliberate “latest wins” policy, but it is not grouping. If all records per station matter, use the explicit grouping loop from Lesson 4.

Before using a dictionary comprehension, ask whether keys are guaranteed unique, whether replacement is intended, and whether overwritten evidence must be preserved.

8. `enumerate()` and `.items()` remain available

Create a lookup from value to first-seen position only when source values are known unique:

```python
codes = ["A1", "B7", "C3"]
positions = {code: index for index, code in enumerate(codes)}
assert positions == {"A1": 0, "B7": 1, "C3": 2}
```

Transform a mapping through its items:

```python
inventory = {"maps": 2, "keys": 0, "torches": 3}
available = {
    item: quantity
    for item, quantity in inventory.items()
    if quantity > 0
}
assert available == {"maps": 2, "torches": 3}
```

The trailing filter checks the current quantity, and the key-value expression preserves accepted pairs.

9. Nested comprehensions follow nested-loop order

A small flattening comprehension mirrors two loops:

```python
rows = [[1, 2], [], [3, 4]]
flat = [value for row in rows for value in row]
assert flat == [1, 2, 3, 4]
```

Read the `for` clauses in the same order as expanded loops:

```python
flat = []
for row in rows:
    for value in row:
        flat.append(value)
```

A small Cartesian product is similar:

```python
pairs = [(left, right) for left in ["A", "B"] for right in [1, 2]]
assert pairs == [("A", 1), ("A", 2), ("B", 1), ("B", 2)]
```

Stop compacting when there are several filters, branching actions, or unfamiliar names. The explicit loop is not inferior; it provides space for intermediate state, rejection evidence, traces, and comments.

10. Comprehension target names do not leak

```python
value = "outside"
squares = [value * value for value in [1, 2, 3]]

assert squares == [1, 4, 9]
assert value == "outside"
```

In modern Python, the comprehension’s target has its own local scope and does not replace the outer `value`. Avoid relying on a comprehension target afterward; use the constructed collection.

An ordinary `for` target behaves differently at top level:

```python
for number in [1, 2, 3]:
    pass

assert number == 3
```

This difference is another reason to keep target names local in meaning and not use them as later results.

11. Do not use a comprehension only for side effects

This creates a list of `None` values merely to print:

```python
messages = ["ready", "launch"]
printed_results = [print(message) for message in messages]
assert printed_results == [None, None]
```

The list is useless because `print()` returns `None`. Use a loop for actions:

```python
for message in messages:
    print(message)
```

Comprehensions build collections. They do not support `break` or `continue`, and they are a poor home for several state changes. Choose an explicit loop for logging, multiple accumulators, early exit, error evidence, or a rule that needs several readable steps.

12. Generator expressions avoid an intermediate collection

A generator expression uses parentheses and supplies values lazily:

```python
values = [3, 5, 8]
has_even = any(value % 2 == 0 for value in values)
assert has_even is True
```

`any()` can stop at the first truthy result, and no full Boolean list is needed. Similarly:

```python
all_positive = all(value > 0 for value in values)
total = sum(value * 2 for value in values)
```

This is a preview, not a full generator lesson. Unit 5 explains iterators, generators, lazy state, and one-pass behavior. For now, recognize the common form when a consuming built-in needs values rather than a reusable list.

Checkpoint: readability and behavior

Comprehension boundaries checkpoint

Choose whether compact collection construction preserves the required behavior.

1. What happens when two inputs create the same key in a dictionary comprehension?

- **A.** The later assignment replaces the earlier value for that key
- **B.** Both values are grouped automatically
- **C.** Python changes the key
- **D.** The result becomes a set

2. How should `[x for row in rows for x in row]` be expanded?

- **A.** Loop over x before rows
- **B.** Loop over each row, then each x in that row
- **C.** Zip all rows
- **D.** Sort rows before reading them

3. Why is `[print(x) for x in values]` usually a poor design?

- **A.** print is forbidden in loops
- **B.** Comprehensions cannot read values
- **C.** It constructs an unnecessary list of print’s None return values

- **D.** It prints only the final value

4. Which task most clearly favors an explicit loop?

- **A.** Double every integer
- **B.** Keep positive values
- **C.** Normalize unique names
- **D.** Collect accepted and rejected records, update totals, and stop on shutdown

Write your choices before checking the answer key at the end of this book.

13. Clean an event log

Use this source log:

```python
events = [
    {"kind": " move ", "value": "N"},
    {"kind": "noise", "value": "?"},
    {"kind": "MOVE", "value": "E"},
    {"kind": "score", "value": 4},
    {"kind": "score", "value": 7},
    {"kind": "move", "value": "N"},
]
```

Build these artifacts with separate, readable comprehensions:

1. `normalized_kinds`: one lowercase stripped kind for every event;
2. `moves`: the move values in source order, accepting kind spellings after normalization;
3. `unique_moves`: a set of accepted move values;
4. `score_by_position`: a dictionary mapping original positions to score values;
5. `latest_by_kind`: a dictionary mapping normalized kind to its latest value;
6. `move_labels`: strings such as `"1: N"` numbered from one; and
7. `high_scores`: score values at least 5.

Run:

```python
assert normalized_kinds == ["move", "noise", "move", "score", "score", "move"]
assert moves == ["N", "E", "N"]
assert unique_moves == {"N", "E"}
assert score_by_position == {3: 4, 4: 7}
assert latest_by_kind == {"move": "N", "noise": "?", "score": 7}
assert move_labels == ["1: N", "2: E", "3: N"]
assert high_scores == [7]
assert events[0]["kind"] == " move "
```

Then expand any two comprehensions into loops and prove the results are equal. Explain why `latest_by_kind` loses earlier values and whether that is acceptable for its stated contract.

Hint: keep normalization beside each filter or output that needs it

Use `event["kind"].strip().lower()` as the normalized expression. Use `enumerate(events)` for original positions and `enumerate(moves, start=1)` for labels. A dictionary comprehension naturally applies “latest value wins” when normalized keys repeat.

Show one complete solution after attempting the lab

PYTHON

```
events = [
    {"kind": " move ", "value": "N"},
    {"kind": "noise", "value": "?"},
    {"kind": "MOVE", "value": "E"},
    {"kind": "score", "value": 4},
    {"kind": "score", "value": 7},
    {"kind": "move", "value": "N"},
]

normalized_kinds = [event["kind"].strip().lower() for event in events]
moves = [
    event["value"]
    for event in events
    if event["kind"].strip().lower() == "move"
]
unique_moves = {move for move in moves}
score_by_position = {
    position: event["value"]
    for position, event in enumerate(events)
    if event["kind"].strip().lower() == "score"
}
latest_by_kind = {
    event["kind"].strip().lower(): event["value"]
    for event in events
}
move_labels = [
    f"{number}: {move}"
    for number, move in enumerate(moves, start=1)
]
high_scores = [
    event["value"]
    for event in events
    if event["kind"].strip().lower() == "score" and event["value"] >= 5
]
```

The expressions are independent so each artifact has one clear job. Repeated normalized keys deliberately leave only the latest event in `latest_by_kind`.

14. Explain the compact forms

1. How do you translate a comprehension into an explicit loop?
2. Why does a trailing filter produce fewer outputs while a conditional expression produces one per input?
3. What happens when transformed set values or dictionary keys collide?
4. Why should code use the resulting collection rather than a target name after a comprehension?
5. Which kinds of state changes or exits are clearer in an explicit loop?

Key points

Key points

- A comprehension builds one new list, set, or dictionary from source traversal.

- A trailing `if` filters inputs; a leading conditional expression chooses one of two outputs for every accepted input.
- Sets remove equal outputs; repeated dictionary keys keep the latest assigned value rather than grouping automatically.
- Expand nested or surprising comprehensions into loops to verify evaluation order.
- Use comprehensions for readable collection construction, not side effects, several accumulators, or early exit.
- A generator expression can feed `any()`, `all()`, or `sum()` without building an intermediate list; deeper lazy behavior belongs to Unit 5. :::

References

- Python tutorial: list comprehensions
- Python tutorial: nested list comprehensions
- Python expression reference: displays for lists, sets, and dictionaries
- Python expression reference: generator expressions

Unit Challenge: Escape the Clockwork Maze

Online lesson and runnable notebook

Guide Pip through a clockwork maze by combining branch precedence, bounded while-loop progress, nested grid scanning, loop control, state collections, and a final comprehension.

- **Level:** Python Foundations · Unit 4 challenge
- **Estimated time:** 60–120 minutes
- **Outcome assessed:** Build complete decisions and terminating loops that scan a grid, reject unusable commands, preserve state evidence, and stop for the correct reason.
- **Evidence:** Twenty-four progressive checks, an escaped-maze artifact, three boundary reruns, and one debugging record

1. Challenge outcome

Pip, a palm-sized maintenance robot, wakes inside a clockwork maze. The exit is visible, but its brass door opens only after Pip collects the winding key. A scratched command tape includes useful moves, an unknown instruction, and one move directly into a wall.

Build the controller that scans the maze, follows the tape, spends energy, collects the key, records every event, and stops as soon as Pip escapes. The challenge is solved when the assertions pass unchanged and the final derived artifact says:

```text
PIP ESCAPED THE CLOCKWORK MAZE
```

Maze rule

Run one assertion group at a time. Repair the first failed promise before moving on. Do not edit expected values, delete awkward commands, hard-code coordinates, or assign the final artifact as one ready-made string. :::

2. Understand the acceptance example

The map uses these tiles:

Tile	Meaning
#	wall; Pip cannot enter
.	open floor
S	starting coordinate
K	winding key
E	exit; escape succeeds only with the key

Tile	Meaning
!	gear hazard; entering it costs one additional energy

The command tape contains `NORTH`, `SOUTH`, `WEST`, and `EAST`. Any other command is unknown. Every command read—including unknown and blocked commands—uses one unit of energy. This guarantees visible progress even when Pip does not move.

Controller contracts

1. Use nested `enumerate()` loops to count terrain and discover `start`, `key_position`, and `exit_position`; do not type their coordinates into those result names.
2. The main simulation is a `while` loop controlled by remaining commands, positive energy, and `not escaped`.
3. Advance `command_index` and reduce energy before any `continue` path.
4. Translate known commands with one complete branch chain.
5. Log an unknown command, then continue without calculating a candidate cell.
6. Use short-circuit evaluation so a row is proven in bounds before its length is read.
7. Log blocked candidates, then continue without changing position.
8. Append every successfully entered coordinate to both an ordered `visited` list and a membership-oriented `visited_set`.
9. Stop immediately when Pip enters `E` with the key.
10. Build path tiles and final artifact words from comprehensions after the loop.

Scope limits

- Keep the supplied public variable names.
- Do not use functions, recursion, exceptions, classes, files, randomness, or third-party packages.
- Do not mutate the maze or command tape.
- Use `break` only for a successful escape; exhaustion and energy depletion are normal condition-based stops.
- Keep blocked and unknown commands in `event_log` rather than discarding them.

3. Start from the contract

Copy this starter into a fresh cell:

```python
maze = [
    "#########",
    "#S..#..E#",
    "#.#.#.#.#",
    "#.#K!...#",
    "#.......#",
    "#########",
]

commands = [
```

PYTHON (CONTINUED)

```
    "SCAN",
    "SOUTH",
    "EAST",
    "SOUTH",
    "SOUTH",
    "EAST",
    "EAST",
    "NORTH",
    "SPIN",
    "EAST",
    "EAST",
    "NORTH",
    "NORTH",
    "EAST",
    "EAST",
]

# Stage A: discovered from the nested grid scan.
start = None
key_position = None
exit_position = None
special_tiles = {"S": [], "K": [], "E": []}
terrain_counts = {}

# Stage B: simulation state.
position = None
command_index = 0
energy = 18
has_key = False
escaped = False
exit_reason = None
visited = []
visited_set = set()
event_log = []

# Stage C: derived after the simulation.
path_tiles = None
artifact_words = None
artifact = None
```

4. Build in small stages

Stage A: scan the maze

Use one outer loop for rows and one inner loop for cells. Count every tile. When a cell belongs to `special_tiles`, append its coordinate. Assign the three public coordinate names from those discovered lists only after scanning. The supplied maze has one of each, but use `None` for a missing key so the boundary variation can finish normally.

PYTHON

```
for row_index, row in enumerate(maze):
    for column_index, tile in enumerate(row):
        # Count terrain and record S, K, and E coordinates here.
        pass

# Derive start, key_position, and exit_position from special_tiles.
```

Initialize simulation position and both visit collections with the discovered start. The list records route order; the set records whether a coordinate has already been entered.

Stage B: process one command per iteration

Use this loop skeleton. Replace comments with the branch and state operations from the contract:

```python
while command_index < len(commands) and energy > 0 and not escaped:
    command = commands[command_index]
    command_index += 1
    energy -= 1

    # Translate a known command into row_delta and column_delta.
    # For an unknown command, append an event and continue.

    # Calculate next_row and next_column from position and the deltas.

    # Prove the row is safe before reading maze[next_row]'s length.
    # For an out-of-bounds or wall candidate, log it and continue.

    # Move, record the coordinate, then handle K, !, and E tiles.
```

Every event begins with the one-based command number, which is the updated `command_index`. Use these exact message shapes:

```text
1: unknown SCAN
3: blocked at (2, 2)
8: key collected at (3, 3)
15: exit reached at (1, 7)
```

Ordinary successful movement uses `"N: moved to (row, column)"`. If Pip reaches the exit without the key, use `"N: exit locked at (row, column)"` and keep processing. Entering `!` consumes one additional energy and logs `"N: hazard at (row, column)"`. After the loop choose `exit_reason` in this precedence: escaped, energy depleted, then commands exhausted.

Stage C: derive the route evidence and artifact

Use a list comprehension over `visited` to create `path_tiles`. Use another comprehension to uppercase these supplied words, then join them:

```python
message_parts = ["Pip", "escaped", "the", "clockwork", "maze"]
```

Do not type the final uppercase sentence as the value of `artifact`.

5. Run progressive assertions

Run each group only after the previous group passes.

Stage A checks: the scan found one of each special tile

```python
assert start == (1, 1)
assert key_position == (3, 3)
assert exit_position == (1, 7)
assert special_tiles == {"S": [(1, 1)], "K": [(3, 3)], "E": [(1, 7)]}
assert terrain_counts == {"#": 31, "S": 1, ".": 19, "E": 1, "K": 1, "!": 1}
```

Stage B checks: the loop progressed and stopped for success

```python
assert command_index == 15
assert energy == 2
assert position == exit_position
assert escaped is True
assert has_key is True
assert exit_reason == "escaped"
```

Event checks: awkward commands remained visible

```python
assert len(event_log) == 15
assert event_log[0] == "1: unknown SCAN"
assert event_log[2] == "3: blocked at (2, 2)"
assert event_log[7] == "8: key collected at (3, 3)"
assert event_log[-1] == "15: exit reached at (1, 7)"
```

Route checks: order and membership tell compatible stories

```python
assert visited == [
    (1, 1),
    (2, 1),
    (3, 1),
    (4, 1),
    (4, 2),
    (4, 3),
    (3, 3),
    (3, 4),
    (3, 5),
    (2, 5),
    (1, 5),
    (1, 6),
    (1, 7),
]
assert visited[0] == start
assert visited[-1] == exit_position
assert visited_set == set(visited)
assert len(visited_set) == 13
```

Stage C checks: every final value was derived

```python
assert path_tiles == ["S", ".", ".", ".", ".", ".", "K", "!", ".", ".", ".", ".", "E"]
assert artifact_words == ["PIP", "ESCAPED", "THE", "CLOCKWORK", "MAZE"]
assert artifact == "PIP ESCAPED THE CLOCKWORK MAZE"
```

That is twenty-four assertions. Do not proceed while an earlier group is failing.

6. Use the hint ladder only when needed

Hint 1: discover and initialize the state

Inside the nested scan, update `terrain_counts` with `.get(tile, 0) + 1`. Membership in `special_tiles` identifies the three special tile kinds. Append `(row_index, column_index)` to the matching list. Each should hold one coordinate, so the public position can select index zero. Initialize `position`, `visited`, and `visited_set` from `start` before the `while` loop.

Hint 2: make progress before every early exit

Read the current command, increment `command_index`, and reduce energy at the top of the loop. An `if`/`elif` chain maps four known strings to `(row_delta, column_delta)`. Its `else` logs the unknown message and continues. For a candidate, use `0 <= next_row < len(maze) and 0 <= next_column < len(maze[next_row])` so the second length lookup is short-circuit protected.

Hint 3: preserve movement before handling special tiles

After a candidate passes bounds and wall checks, update `position`, append it to `visited`, and add it to `visited_set`. Inspect the entered tile. The key branch sets `has_key`; the hazard branch spends one more energy; the exit-with-key branch sets `escaped`, logs success, and breaks; the exit-without-key branch logs a locked door. Derive `exit_reason` only after the loop.

7. Keep debugging evidence

Retain one complete record from a real failure:

Field	Your evidence
First failing assertion	Paste the exact assertion and observed values.
Current state	Record command number, command, position, energy, and latest event.
Hypothesis	Name one branch, update, or stopping rule that could explain it.
Controlled change	Describe the one change you made.
Clean rerun	Record the first assertion group that passed afterward.

Useful temporary trace:

```python
print(
    "command:", command_index,
    "position:", position,
    "energy:", energy,
    "key:", has_key,
    "latest:", event_log[-1],
)
```

Remove or disable the trace after the checks pass. Keep the written evidence.

8. Test three changed mazes

Restart from the original starter before each variation.

1. **Missing key:** replace `K` with `.`. Predict whether Pip reaches the exit, whether it escapes, and whether the loop ends through exhaustion. Update only assertions whose promised behavior changes.
2. **Low energy:** start with energy 7. Predict the final command index, position, and exit reason before running.
3. **More invalid input:** insert `"JUMP"` immediately before the final `EAST`. Prove it appears in the event log and consumes energy without changing the successful route.

For each variation, record the first condition that prevents another iteration or the exact `break` that stops it.

After predicting, compare your results with this boundary table and write one assertion for each listed field:

Variation	Command index	Final position	Energy	Key	Escaped	Exit reason
key replaced by `.`	15	`(1, 7)`	2	false	false	`commands exhausted`
starting energy 7	7	`(4, 3)`	0	false	false	`energy depleted`
`JUMP` before final EAST	16	`(1, 7)`	1	true	true	`escaped`

9. Compare with a solution path

Reveal after your checks pass or all three hints have been used

PYTHON

```
maze = [
    "#########",
    "#S..#..E#",
    "#.#.#.#.#",
    "#.#K!...#",
    "#.......#",
    "#########",
]
commands = [
    "SCAN", "SOUTH", "EAST", "SOUTH", "SOUTH",
    "EAST", "EAST", "NORTH", "SPIN", "EAST",
    "EAST", "NORTH", "NORTH", "EAST", "EAST",
]

special_tiles = {"S": [], "K": [], "E": []}
terrain_counts = {}
for row_index, row in enumerate(maze):
    for column_index, tile in enumerate(row):
        terrain_counts[tile] = terrain_counts.get(tile, 0) + 1
        if tile in special_tiles:
            special_tiles[tile].append((row_index, column_index))

start = special_tiles["S"][0]
key_position = special_tiles["K"][0] if special_tiles["K"] else None
exit_position = special_tiles["E"][0]

position = start
command_index = 0
energy = 18
has_key = False
escaped = False
exit_reason = None
visited = [start]
visited_set = {start}
event_log = []

while command_index < len(commands) and energy > 0 and not escaped:
    command = commands[command_index]
```

PYTHON (CONTINUED)

```
    command_index += 1
    energy -= 1

    if command == "NORTH":
        row_delta, column_delta = -1, 0
    elif command == "SOUTH":
        row_delta, column_delta = 1, 0
    elif command == "WEST":
        row_delta, column_delta = 0, -1
    elif command == "EAST":
        row_delta, column_delta = 0, 1
    else:
        event_log.append(f"{command_index}: unknown {command}")
        continue

    next_row = position[0] + row_delta
    next_column = position[1] + column_delta
    in_bounds = (
        0 <= next_row < len(maze)
        and 0 <= next_column < len(maze[next_row])
    )

    if not in_bounds or maze[next_row][next_column] == "#":
        event_log.append(
            f"{command_index}: blocked at {(next_row, next_column)}"
        )
        continue

    position = (next_row, next_column)
    visited.append(position)
    visited_set.add(position)
    tile = maze[next_row][next_column]

    if tile == "K" and not has_key:
        has_key = True
        event_log.append(f"{command_index}: key collected at {position}")
    elif tile == "!":
        energy -= 1
        event_log.append(f"{command_index}: hazard at {position}")
    elif tile == "E" and has_key:
        escaped = True
        event_log.append(f"{command_index}: exit reached at {position}")
        break
    elif tile == "E":
        event_log.append(f"{command_index}: exit locked at {position}")
    else:
        event_log.append(f"{command_index}: moved to {position}")

if escaped:
    exit_reason = "escaped"
elif energy <= 0:
    exit_reason = "energy depleted"
else:
    exit_reason = "commands exhausted"

path_tiles = [maze[row][column] for row, column in visited]
message_parts = ["Pip", "escaped", "the", "clockwork", "maze"]
artifact_words = [word.upper() for word in message_parts]
artifact = " ".join(artifact_words)

print(artifact)
```

The command index and energy change before either `continue`, so unknown and blocked commands cannot trap the controller. The bounds expression protects the row lookup, and the ordered list plus set preserve two different route questions.

10. Check your understanding

Answer these questions before recording the challenge. The quiz runs directly in your browser.

Clockwork-maze control decisions

Choose the rule that keeps Pip’s controller complete and terminating.

1. Why are command index and energy updated before an unknown-command `continue`?

- **A.** Every iteration path must consume input and move toward stopping
- **B.** continue updates both names automatically
- **C.** Unknown commands should move Pip
- **D.** The maze is immutable

2. Why check the candidate row bounds before `len(maze[next_row])`?

- **A.** Rows cannot contain strings
- **B.** Short-circuiting skips the unsafe row lookup when the row is invalid
- **C.** len() moves Pip
- **D.** The column must always be zero

3. Why preserve both `visited` and `visited_set`?

- **A.** Sets preserve duplicate route order
- **B.** Lists provide faster membership in every case
- **C.** The list records route order while the set represents unique membership
- **D.** Python requires collections in pairs

4. When is `break` justified in the simulation?

- **A.** After every blocked move
- **B.** After every unknown command
- **C.** As soon as any key is visible
- **D.** After Pip enters the exit with the key, because later commands cannot improve success

Write your choices before checking the answer key at the end of this book.

11. Decide whether the challenge is complete

Evidence rubric

Evidence	Ready to record when
Behavior	All twenty-four supplied assertions pass unchanged.
Control flow	You can identify the selected branch, every progress update, each `continue`, and the successful `break`.
State	The ordered route, unique visited coordinates, event log, energy, and exit reason agree.
Boundaries	Missing-key, low-energy, and additional-invalid-command reruns match written predictions.
Debugging	One failure record connects observed state, one hypothesis, one change, and a clean rerun.
Reproducibility	The original and each variation work from separately initialized clean state.

Record completion only when every statement is true:

- ☐ All supplied assertions pass without editing expected values.
- ☐ Coordinates and the final message are derived rather than hard-coded.
- ☐ The work runs from a clean notebook state.
- ☐ I can explain why every possible while-loop path progresses or exits.
- ☐ I kept one complete debugging record.
- ☐ I predicted and verified all three variations.

This button stores a self-reported marker only in this browser. It does not submit the artifact, grade it, verify identity, or issue a certificate.

Key points

Key points

- Scan structured data instead of hard-coding facts the data already contains.
- Make input, budget, or state progress before a path can continue.
- Order guards so short-circuiting protects operations that are not always safe.
- Keep ordered evidence, membership evidence, and event explanations in the collection shapes that preserve their meaning.
- Stop as soon as the promised success cannot be improved, and preserve an exit reason for every other outcome. :::

Unit 5: Building Reusable Functions

Online lesson and runnable notebook

Turn working control-flow scripts into named, reusable tools with clear inputs, returned results, controlled effects, and call behavior you can trace.

- **Level:** Python Foundations · Unit 5
- **Estimated time:** 45–75 minutes for orientation and readiness work
- **You will learn:** Read a function as a contract, trace values through calls, and follow the eight-lesson path from ordinary return values to recursive, lazy, and decorated behavior.
- **Practice in:** Google Colab, JupyterLab, or a local editor

1. A long script becomes a set of named tools

Unit 4 gave you the control flow to summarize expedition distances:

PYTHON
```
distances = [4, 7, 3]
total = 0

for distance in distances:
    total += distance

message = f"Route total: {total} km"
print(message)
```

That script works once. A function turns the calculation into a tool that other code can call with different routes:

PYTHON
```
def total_distance(distances):
    """Return the sum of an expedition's distances."""
    total = 0
    for distance in distances:
        total += distance
    return total


day_one = total_distance([4, 7, 3])
day_two = total_distance([2, 5])

assert day_one == 14
assert day_two == 7
```

The loop still performs the calculation. The new function boundary answers additional design questions:

- What input must the caller provide?
- What result returns to the caller?
- Which names exist only during one call?

- Does the function change anything outside itself?
- Can another function receive, return, pause, or wrap this behavior?

Those questions—not the number of lines—make functions a design tool.

2. Read every function as a contract

Use this compact contract before examining the body:

Contract part	Question	`total_distance` answer
Purpose	What job has this name?	total route distances
Parameters	What must the caller supply?	an iterable of numeric distances
Returned result	What crosses back to the caller?	one numeric total
Side effects	What observable state changes?	none intended
Boundary	What happens for empty input?	returns `0`

A docstring and annotations may communicate parts of a contract, but the body must actually honor it. Assertions provide executable examples:

```python
assert total_distance([]) == 0
assert total_distance([5]) == 5
```

A contract is not a promise that every possible value is valid. It identifies the supported inputs and behavior clearly enough that callers can use the function and check important boundaries.

3. Trace the call, not only the final output

When Python evaluates this line:

```python
label = f"Distance: {total_distance([4, 7])} km"
```

several moments occur:

Moment	Active code	Important state
1	caller	builds argument list `[4, 7]`
2	function call	binds that list to local parameter `distances`
3	function body	creates and updates local `total`
4	`return`	sends 11 to the suspended caller
5	caller resumes	formats `"Distance: 11 km"` and binds `label`

Every call receives its own local frame. Later lessons add enclosing scopes, recursive stacks, suspended generator frames, and wrapper calls, but the same questions remain useful: what entered, what executed, what changed, and what returned where?

4. Put behavior in a function for a reason

A function is usually helpful when it provides at least one of these benefits:

- several callers need the same behavior;
- a meaningful name makes the program easier to read;
- one responsibility can be checked independently;
- a changing policy can be supplied as a parameter or callback;
- a complex script becomes a small composition of explicit steps; or
- local state should belong to one call rather than the entire program.

Do not create a function only to make every block shorter. This adds a name without adding meaning:

PYTHON

```
def add_one(number):
    return number + 1
```

It may be useful if “advance one game level” is a real reusable rule. It is noise if it merely hides an obvious operator used once. Design from responsibility and contract, not an arbitrary line limit.

5. Your path through this unit

Lesson	Capability you will build	Main lab artifact
1	Defining Functions and Returning Useful Results — separate definition from execution, return reusable values, and handle every intended path.	Expedition summary
2	Designing Clear Parameters and Calls — predict argument binding and design safe positional, keyword, default, and variadic interfaces.	Mission-message builder
3	Breaking a Program into Small Functions — compose cohesive calculations while keeping printing, mutation, and dependencies explicit.	Expedition report pipeline
4	Following Names Through Function Calls — trace frames, LEGB lookup, return points, closures, and `nonlocal` state.	Dispatch-center trace
5	Passing Functions as Values — configure reusable processing with callbacks, key functions, factories, and small lambdas.	Tournament leaderboard
6	Solving Smaller Problems with Recursion — define base/progress cases and trace results through a nested data shape.	Nested quest archive
7	Producing Values One at a Time — explain one-pass consumption and build finite lazy pipelines with `yield`.	Sensor stream
8	Adding Behavior with Decorators — wrap functions while preserving calls, results, metadata, and stacking order.	Achievement tracker

The Unit Challenge asks you to unlock a nested Moonlit Library with cohesive functions, a recursive generator, callback selection, and an audited unlock operation.

6. Unit 4 supplied the bodies

You already know how to:

- choose complete branches;
- transform, filter, total, count, group, and search in loops;

- explain `while` progress and stopping;
- traverse nested data; and
- build readable comprehensions.

Unit 5 places those algorithms behind explicit call boundaries. The function body is not a new universe: parameters become its starting values, local names hold its working state, and `return` makes its product available to the caller.

```python
def accepted_readings(readings):
    accepted = []
    for reading in readings:
        if reading >= 0:
            accepted.append(reading)
    return accepted


source = [8, -2, 0, 5]
assert accepted_readings(source) == [8, 0, 5]
assert source == [8, -2, 0, 5]
```

7. Unit 6 will deepen object sharing

This unit identifies whether a function mutates an input and generally builds separate result collections when that keeps the contract simple. Unit 6 develops the full model of identity, aliases, shallow and deep copies, hashability, and mutation across function boundaries.

For now, state the effect plainly:

```python
def add_stamp(stamps):
    stamps.append("approved")  # Mutates the supplied list.


def with_stamp(stamps):
    return stamps + ["approved"]  # Returns a new outer list.
```

Neither shape is automatically wrong. The caller must be able to tell which contract applies.

8. Set up a call laboratory

Use separate cells or sections for definitions, source inputs, calls, and checks:

```python
# 1. Definitions
def route_status(stops):
    if not stops:
        return "no route"
    return f"{len(stops)} stops"


# 2. Source inputs
route = ["dock", "ridge", "clinic"]

# 3. Calls and observable results
status = route_status(route)

# 4. Checks
assert status == "3 stops"
```

PYTHON (CONTINUED)

```
assert route == ["dock", "ridge", "clinic"]
```

Restart and run all before trusting stateful examples. A closure can remember old calls, and an iterator can remain partly consumed even when the source code looks unchanged. Clean initialization separates a real rule from leftover notebook state.

9. Choose a realistic pace

Unit 5 is planned for approximately **29–43 hours**, including orientation, examples, modifications, checkpoints, eight final labs, and the challenge. The range is guidance, not a deadline.

A sustainable rhythm is:

1. state the contract before reading the implementation;
2. trace one call through binding, body, and return;
3. change one argument or boundary and update the prediction;
4. answer a checkpoint without copying its nearby sentence;
5. build the final lab before opening support; and
6. rerun stateful closure, recursion, iterator, generator, and decorator examples from clean definitions.

10. Check your readiness

Wrap this Unit 4 calculation in a function named `usable_total`:

PYTHON

```
readings = [8, -2, 0, 5]
total = 0

for reading in readings:
    if reading >= 0:
        total += reading
```

Your function must accept a readings collection, return the total of values at least zero, return `0` for empty input, and leave its argument unchanged.

PYTHON

```
source = [8, -2, 0, 5]

assert usable_total(source) == 13
assert usable_total([]) == 0
assert source == [8, -2, 0, 5]
```

You are ready for Lesson 1 when you can identify the definition, call, argument, parameter, local accumulator, and returned result—and explain why no calculation runs merely because Python reads `def`.

Key points

Key points

- A function gives behavior a name and a call boundary; it is useful when that boundary communicates a real responsibility.
- Read purpose, parameters, returned result, side effects, and boundaries as one contract.
- Trace argument evaluation, binding, local execution, return, and caller resumption separately.
- Unit 4 control flow becomes function-body behavior; Unit 6 will deepen shared object and mutation mechanics.
- Restart stateful examples so old calls or partial consumption cannot imitate correct behavior. :::

Defining Functions and Returning Useful Results

Online lesson and runnable notebook

Define and call functions, trace when bodies execute, return reusable values on every intended path, and distinguish returned data from printed output and implicit None.

- **Level:** Python Foundations · Unit 5
- **Estimated time:** 3.5–5 hours
- **You will learn:** Define reusable behavior, trace definition and call time, and return useful results consistently without confusing display with data flow.
- **Practice in:** Google Colab, JupyterLab, or a local editor

1. Give a repeated calculation one reliable home

An expedition notebook calculates route totals in two places:

```python
day_one = [4, 7, 3]
day_one_total = 0
for distance in day_one:
    day_one_total += distance

day_two = [2, 5]
day_two_total = 0
for distance in day_two:
    day_two_total += distance
```

The two copies currently agree, but a future repair could reach only one. Put the algorithm behind one function boundary:

```python
def total_distance(distances):
    """Return the sum of the supplied distances."""
    total = 0
    for distance in distances:
        total += distance
    return total


day_one_total = total_distance([4, 7, 3])
day_two_total = total_distance([2, 5])

assert day_one_total == 14
assert day_two_total == 7
```

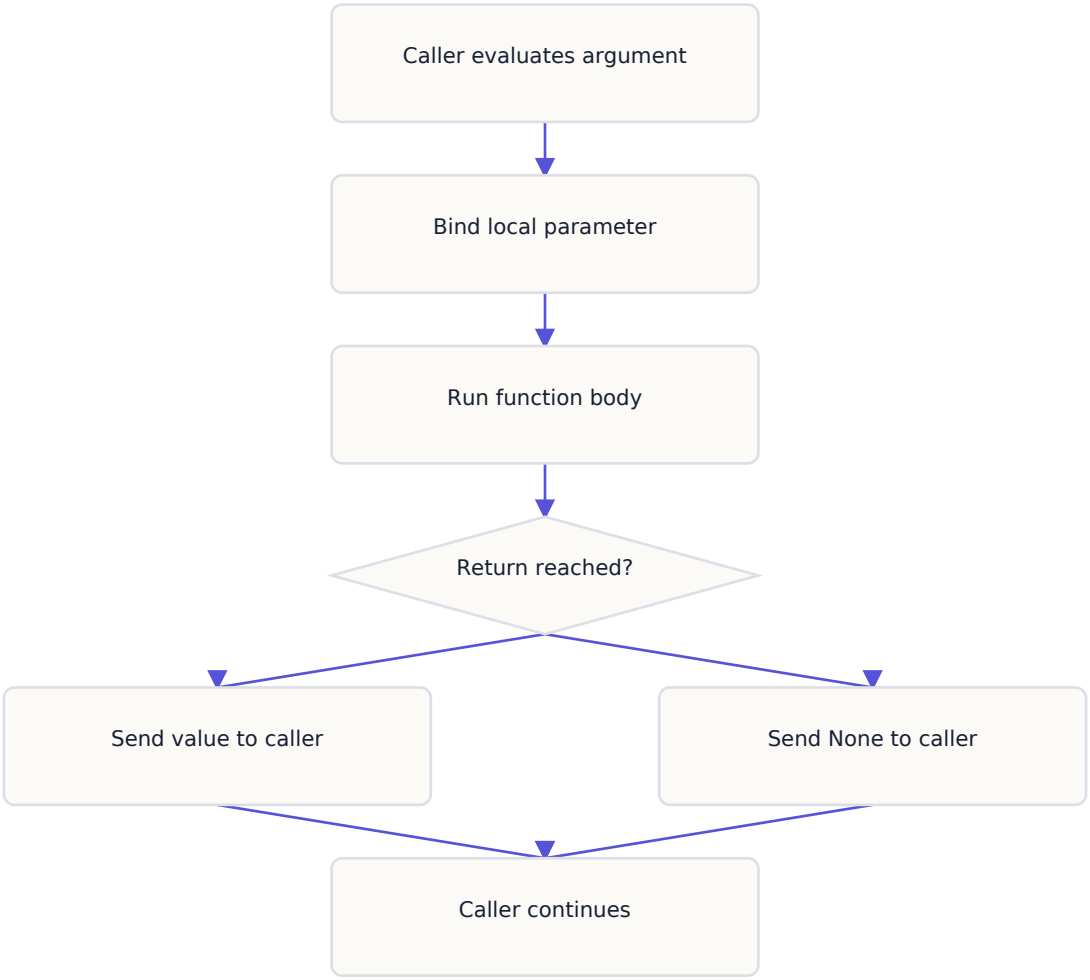
The function has one job. Its parameter gives each call an input, its local accumulator holds working state, and `return` sends the result back.

Questions this lesson will answer

- What happens when Python reads a function definition?
- When does the indented body actually run?

- How is returning a value different from printing it?
- What returns when execution reaches the end without `return`?
- How can every supported branch provide a consistent result?

A call evaluates its argument, creates local parameter state, runs the body, and returns one result to the suspended caller.



Return reached? → Send value to caller: Yes

Return reached? → Send None to caller: No, body ends

2. Defining a function does not run its body

PYTHON

```
def announce_launch():
    print("Launch sequence started")


print("Definition complete")
```

Only `Definition complete` appears. When Python executes the `def` statement, it creates a function object and binds it to `announce_launch`. The indented body is stored as the behavior to run later.

Parentheses request a call:

```python
print("Before call")
announce_launch()
print("After call")
```

The order is:

1. display `Before call`;
2. enter `announce_launch` and display its message;
3. finish the body and return to the caller; and
4. display `After call`.

A blank line between the definition and the call is style, not control flow. Indentation determines the body; alignment returns to top-level code.

3. Parameters receive a fresh call's inputs

```python
def route_label(name):
    return f"Route: {name}"


north_label = route_label("Northern Ridge")
south_label = route_label("Southern Caves")
```

`name` is a **parameter** in the definition. `"Northern Ridge"` and `"Southern Caves"` are **arguments** supplied by separate calls. During the first call, local `name` refers to the first string. That frame finishes before a new frame binds `name` to the second string.

The parameter name need not match the caller's name:

```python
selected_route = "Moon Pass"
label = route_label(selected_route)
assert label == "Route: Moon Pass"
```

Python passes the value produced by evaluating `selected_route`; it does not connect the two names permanently. Lesson 2 develops complete binding rules.

4. Calling and referring to a function are different

```python
def route_count(stops):
    return len(stops)


function_value = route_count
result_value = route_count(["dock", "ridge"])

assert callable(function_value)
assert result_value == 2
```

Without parentheses, `route_count` refers to the function object. With parentheses, Python calls it and the expression becomes its returned result. Lesson 5 uses function objects as configurable behavior. For now, check whether you intend to name the tool or use it.

Checkpoint: definitions and calls

Definition and call checkpoint

Trace when Python creates behavior and when it executes it.

1. What happens when Python executes a `def` statement?

- **A.** It creates a function object and binds its name without running the body
- **B.** It runs the body once and discards it
- **C.** It asks the user for arguments
- **D.** It converts the body to printed text

2. Which expression calls `route_count` with one list argument?

- **A.** `route_count`
- **B.** `route_count(['dock'])`
- **C.** `def route_count`
- **D.** `['dock'].route_count`

3. In `def label(name):` followed by `label('Ada')`, which statement is correct?

- **A.** Both are parameters
- **B.** Both are arguments
- **C.** `name` is the parameter and `'Ada'` is the argument
- **D.** `'Ada'` is the return statement

4. What does `tool = route_count` store?

- **A.** The count from an empty route
- **B.** The source list
- **C.** None
- **D.** The function object without calling it

Write your choices before checking the answer key at the end of this book.

5. A return value remains useful to the caller

PYTHON

```
def distance_status(total):
    if total >= 20:
        return "long route"
    return "short route"


status = distance_status(14)
assert status == "short route"
```

`return` performs two actions:

1 it ends the current call; and

2 it makes the return expression’s value become the value of the call expression.

That value can participate in later work:

```python
assert distance_status(25) == "long route"

statuses = [distance_status(14), distance_status(25)]
message = f"First expedition: {distance_status(14)}"

assert statuses == ["short route", "long route"]
assert message == "First expedition: short route"
```

A function’s result is more reusable than one hard-coded display.

6. Printing is an effect; returning is data flow

This function displays a calculation but does not return it:

```python
def show_total(distances):
    print(sum(distances))


received = show_total([4, 7, 3])
print("Caller received:", received)
```

The function prints `14`, then the caller displays `Caller received: None`. `print()` sends a display representation to a human. Its own return value is `None`, and reaching the end of `show_total` also returns `None` implicitly.

A calculation function should normally return the number:

```python
def calculate_total(distances):
    return sum(distances)


received = calculate_total([4, 7, 3])
assert received == 14
```

The caller chooses whether to display, compare, store, or format it:

```python
print(f"Total: {received} km")
```

Command-style functions can intentionally return `None` when their contract is an effect. The problem is not `None`; the problem is promising useful data and then only printing it.

7. Falling off the end returns `None`

```python
def find_first_long_route(routes):
    for route in routes:
        if route["distance"] >= 20:
            return route


routes = [{"name": "ridge", "distance": 14}]
result = find_first_long_route(routes)
assert result is None
```

No route matches. The loop finishes and the body ends without an explicit `return`, so the call produces `None`.

Make the boundary visible:

```python
def find_first_long_route(routes):
    for route in routes:
        if route["distance"] >= 20:
            return route
    return None
```

The behavior is the same, but the explicit line communicates that no match is an expected result rather than an accidental omission.

Checkpoint: return values and `None`

Return value checkpoint

Distinguish displayed output from the value received by the caller.

1. What does `return value` do?

- **A.** Ends the current call and sends the value to its caller
- **B.** Only displays the value
- **C.** Restarts the function
- **D.** Makes every local name global

2. A function only calls `print(14)` and reaches its end. What does `result = that_function()` bind to?

- **A.** The string `'14'`
- **B.** None
- **C.** The print function
- **D.** A list containing 14

3. Why might a search function end with explicit `return None`?

- **A.** To call the function again
- **B.** To convert all records
- **C.** To communicate that no match is an expected result

- **D.** To print a blank line

4. Which design lets the caller store, compare, and later format a total?

- **A.** Print it and return nothing
- **B.** Use only a comment
- **C.** Bind it to a local name but end without return
- **D.** Return the numeric total

Write your choices before checking the answer key at the end of this book.

8. `return` stops the current call immediately

```python
def boarding_message(has_ticket, gate_open):
    if not has_ticket:
        return "ticket required"
    if not gate_open:
        return "gate closed"
    return "welcome aboard"


assert boarding_message(False, True) == "ticket required"
assert boarding_message(True, False) == "gate closed"
assert boarding_message(True, True) == "welcome aboard"
```

The first satisfied guard returns immediately. Later conditions are not evaluated in that call. This flat structure can be clearer than nesting every successful step.

Statements after an unconditional return in the same path are unreachable:

```python
def doubled(number):
    return number * 2
    print("This line cannot run")
```

Python accepts the syntax, but execution leaves before the `print()`. Remove unreachable code rather than trusting indentation to make it run.

9. Every supported path should honor one result contract

This function mixes a string with an accidental `None`:

```python
def temperature_label(value):
    if value < 0:
        return "freezing"
    if value > 30:
        return "hot"


assert temperature_label(10) is None
```

If the contract promises a label for every numeric temperature, add the missing middle result:

```python
def temperature_label(value):
    if value < 0:
        return "freezing"
    if value > 30:
        return "hot"
    return "mild"


assert temperature_label(-1) == "freezing"
assert temperature_label(10) == "mild"
assert temperature_label(31) == "hot"
```

A deliberate sentinel may still be correct when the domain defines “no result.” Consistency means every path matches the stated contract, not that every function must return a string.

10. Return related values together

A function can return a tuple:

```python
def route_summary(distances):
    total = sum(distances)
    count = len(distances)
    return total, count


total, count = route_summary([4, 7, 3])
assert (total, count) == (14, 3)
```

The comma constructs one tuple result, and the caller unpacks its two positions. Use this for a small, obvious fixed group. A dictionary with named fields may be clearer when there are many results or their roles are easy to confuse.

11. Call expressions evaluate from the inside out

```python
def add_tax(price):
    return price * 1.08


def format_price(price):
    return f"${price:.2f}"


label = format_price(add_tax(10))
assert label == "$10.80"
```

Python must know the argument for `format_price`, so it first calls `add_tax(10)`. That returns `10.8`; then `format_price(10.8)` runs. Expanded names expose the same order:

```python
taxed_price = add_tax(10)
label = format_price(taxed_price)
```

Prefer intermediate names when the values matter during explanation or debugging.

Checkpoint: paths and call order

Return paths and call order checkpoint

Trace which statements execute and what one call ultimately produces.

1. After a return statement executes, what happens to later statements in that call path?

- **A.** They are skipped because the call has ended
- **B.** They run before the return value leaves
- **C.** They become global
- **D.** They run on the next call automatically

2. A label function returns for values below 0 and above 30 but has no final return. What does a value of 10 produce?

- **A.** `'mild'` automatically
- **B.** None
- **C.** Zero
- **D.** The original argument

3. What does `return total, count` return?

- **A.** Two independent calls
- **B.** Only total
- **C.** One tuple containing the two values
- **D.** A printed sentence

4. In `format_price(add_tax(10))`, which function body runs first?

- **A.** format_price
- **B.** Both simultaneously
- **C.** Neither until print
- **D.** add_tax

Write your choices before checking the answer key at the end of this book.

12. Build an expedition summary

Complete the contract before opening support:

```python
def summarize_route(distances):
    """Return total distance, stop count, and route status."""
    pass


source_route = [4, 7, 3]
summary = summarize_route(source_route)
```

The function must:

1. accept one collection of non-negative numeric distances;
2. return a three-item tuple `(total, count, status)`;
3. return status `"no route"` for empty input;
4. otherwise return `"short route"` below 20 total and `"long route"` at least 20;
5. use early return for the empty case;
6. not print or mutate the source; and
7. leave no supported path with an implicit result.

Run:

```python
assert summary == (14, 3, "short route")
assert summarize_route([]) == (0, 0, "no route")
assert summarize_route([20]) == (20, 1, "long route")
assert summarize_route([0]) == (0, 1, "short route")
assert source_route == [4, 7, 3]
```

Then use the returned tuple to build this display **outside** the function:

```text
3 stops | 14 km | short route
```

Hint: decide the complete result at each return point

Handle empty input first with one explicit tuple. For non-empty input, calculate total and count, choose status, and return all three values together. Printing belongs to the caller.

Show one complete solution after attempting the lab

```python
def summarize_route(distances):
    """Return total distance, stop count, and route status."""
    if not distances:
        return 0, 0, "no route"

    total = sum(distances)
    count = len(distances)
    if total >= 20:
        status = "long route"
    else:
        status = "short route"
    return total, count, status


source_route = [4, 7, 3]
summary = summarize_route(source_route)
total, count, status = summary
display = f"{count} stops | {total} km | {status}"
```

The caller owns presentation, so the returned numbers remain useful for later comparisons or calculations.

13. Explain the call boundary

1. What does Python do at the definition, and what waits until a call?

2 Why is printed output not the return value received by the caller?

3 When is implicit `None` deliberate, and when does it reveal a missing path?

4 Why can no statement after an executed `return` affect that call?

5 In a nested call, why must the inner result exist before the outer body begins?

Key points

Key points

- Executing `def` creates and binds a function; calling the function runs its body.
- Parameters are local input names, while arguments are values evaluated by the caller.
- `return` ends one call and supplies the call expression’s value.
- Printing is an observable effect, not a substitute for returning reusable data.
- Reaching the end returns `None`; make expected no-result behavior explicit.
- Every supported path should honor one stated result contract.
- Small tuples can return several closely related results; caller code can unpack and present them. :::

References

- Python tutorial: defining functions
- Python language reference: function definitions
- Python language reference: the `return` statement
- Python built-in function: `callable`

Designing Clear Parameters and Calls

Online lesson and runnable notebook

Predict how Python binds positional, keyword, default, keyword-only, variadic, and unpacked arguments while designing interfaces that remain readable across repeated calls.

- **Level:** Python Foundations · Unit 5
- **Estimated time:** 4–5.5 hours
- **You will learn:** Read function signatures, predict argument binding and call errors, use safe defaults and keyword-only configuration, and unpack or forward arguments deliberately.
- **Practice in:** Google Colab, JupyterLab, or a local editor

1. Make configurable calls readable

This call is legal but difficult to interpret:

```python
def mission_message(name, distance, precision, unit, urgent):
    prefix = "URGENT: " if urgent else ""
    return f"{prefix}{name}: {distance:.{precision}f} {unit}"


message = mission_message("Moon Pass", 12.345, 1, "km", True)
```

Which `1` controls precision? What does `True` mean? A clearer interface separates essential data from configuration:

```python
def mission_message(
    name,
    distance,
    *,
    precision=1,
    unit="km",
    urgent=False,
):
    prefix = "URGENT: " if urgent else ""
    return f"{prefix}{name}: {distance:.{precision}f} {unit}"


message = mission_message("Moon Pass", 12.345, urgent=True)
assert message == "URGENT: Moon Pass: 12.3 km"
```

`name` and `distance` may be supplied by position. Parameters after `*` are keyword-only, so their meaning appears at every call.

Questions this lesson will answer

- When does Python evaluate arguments, and how are they bound?
- Which call shapes raise `TypeError` before the body starts?
- Why can a mutable default remember earlier calls?

- When do `*args`, `**kwargs`, `*sequence`, and `**mapping` help?
- What do docstrings and annotations promise—and what do they not enforce?

2. Python evaluates arguments before the body begins

```python
def announce(value):
    print("body received", value)
    return value


def prepare():
    print("argument prepared")
    return "map"


result = announce(prepare())
```

The output order is:

```text
argument prepared
body received map
```

Python evaluates the argument expression `prepare()` first. Only after it returns `"map"` can Python bind local parameter `value` and begin `announce`.

Multiple argument expressions are evaluated left to right:

```python
def mark(label):
    print(label)
    return label


def pair(left, right):
    return left, right


result = pair(mark("left"), mark("right"))
assert result == ("left", "right")
```

Do not put surprising state changes inside arguments. The evaluation order is predictable, but explicit intermediate names are often clearer.

3. Positional arguments bind by order

```python
def describe_stop(name, distance):
    return f"{name} is {distance} km away"


assert describe_stop("Ridge", 7) == "Ridge is 7 km away"
```

The first argument binds to `name`; the second binds to `distance`. Reversing them does not produce a binding error—it produces a nonsensical message:

```python
assert describe_stop(7, "Ridge") == "7 is Ridge km away"
```

Python binds by position, not intended meaning. Useful parameter names, annotations, and keyword calls help humans detect the mistake.

4. Keyword arguments bind by parameter name

```python
assert describe_stop(distance=7, name="Ridge") == "Ridge is 7 km away"
```

Keyword arguments can appear in a different order because their labels determine binding. Positional arguments must come before ordinary keyword arguments in a call:

```python
assert describe_stop("Ridge", distance=7) == "Ridge is 7 km away"
```

A value cannot bind the same parameter twice:

```python
def describe_stop(name, distance):
    return f"{name} is {distance} km away"


# describe_stop("Ridge", name="Cave", distance=7)
```

If uncommented, Python raises `TypeError` because positional `"Ridge"` already bound `name`, then `name="Cave"` tries again. Binding fails before the body runs.

Checkpoint: argument binding

Argument binding checkpoint

Predict parameter bindings before considering the function body.

1. In `announce(prepare())`, when does `prepare()` run?

- **A.** Before announce’s body so its result can become the argument
- **B.** After announce returns
- **C.** Only if announce prints
- **D.** At function definition time

2. Why can `describe_stop(distance=7, name='Ridge')` reverse the signature order?

- **A.** Strings are always first
- **B.** Keyword names determine which parameter receives each value
- **C.** Python sorts every argument
- **D.** distance has a hidden default

3. Why does `f(1, value=2)` fail when the first parameter is named `value`?

- **A.** Integers cannot be arguments

- **B.** Keywords must be uppercase
- **C.** The same parameter receives both a positional and keyword value
- **D.** Function bodies cannot use value

4. What determines binding for two positional arguments?

- **A.** Their runtime types
- **B.** Their variable names in the caller
- **C.** Alphabetical order
- **D.** Their order in the call relative to the signature

Write your choices before checking the answer key at the end of this book.

5. Call-shape mistakes fail before body execution

For this definition:

```
PYTHON
def route_label(name, distance):
    return f"{name}: {distance} km"
```

These commented calls each have a different binding problem:

```
PYTHON
# route_label("Ridge")
# route_label("Ridge", 7, "north")
# route_label("Ridge", miles=7)
```

If run separately, they raise `TypeError` for:

1. a missing required `distance` argument;
2. too many positional arguments; and
3. an unexpected `miles` keyword.

Read the function name and parameter names in the message. Unit 8 develops a full traceback method; here the immediate fact is that no valid local parameter frame could be constructed.

6. Defaults make an input optional for the caller

```
PYTHON
def route_label(name, distance, unit="km"):
    return f"{name}: {distance} {unit}"


assert route_label("Ridge", 7) == "Ridge: 7 km"
assert route_label("Ridge", 7, "miles") == "Ridge: 7 miles"
assert route_label("Ridge", 7, unit="m") == "Ridge: 7 m"
```

The default is used only when the call supplies no value for `unit`. Required ordinary parameters must appear before defaulted ones in the definition.

Defaults are part of the public contract. Changing `unit="km"` to `"miles"` changes existing calls that omitted that argument even though their code has not changed.

7. A mutable default remembers earlier calls

Default expressions run once when Python executes the definition:

```python
def record_stop(stop, log=[]):
    log.append(stop)
    return log


first = record_stop("dock")
second = record_stop("ridge")

assert first == ["dock", "ridge"]
assert second == ["dock", "ridge"]
assert first is second
```

Both calls use the same default list. This may surprise callers expecting fresh per-call storage.

Use `None` as a sentinel and create the list inside each call:

```python
def record_stop(stop, log=None):
    if log is None:
        log = []
    log.append(stop)
    return log


first = record_stop("dock")
second = record_stop("ridge")

assert first == ["dock"]
assert second == ["ridge"]
assert first is not second
```

An explicitly supplied list is still deliberately changed:

```python
shared_log = []
result = record_stop("dock", shared_log)
assert shared_log == ["dock"]
assert result is shared_log
```

Unit 6 develops identity and mutation in depth. The interface lesson’s rule is: never use a mutable default as accidental per-call state.

8. Keyword-only parameters make configuration explicit

```python
def format_distance(distance, *, precision=1, unit="km"):
    return f"{distance:.{precision}f} {unit}"


assert format_distance(12.345) == "12.3 km"
assert format_distance(12.345, precision=2) == "12.35 km"
```

A call such as `format_distance(12.345, 2)` raises `TypeError`; `precision` must be named. This is useful when optional values share types or when the call should read like configuration.

Avoid “Boolean soup”:

```python
def report(name, *, uppercase=False, include_count=True):
    pass
```

`report("ridge", uppercase=True, include_count=False)` communicates far more than `report("ridge", True, False)`.

Checkpoint: defaults and keyword-only inputs

Defaults and keyword-only checkpoint

Trace repeated calls and decide which interface makes configuration visible.

1. When does a function use a parameter's default value?

- **A.** When the caller does not supply that parameter
- **B.** Whenever the argument is zero
- **C.** Only on the first call
- **D.** After the body returns

2. Why can `def f(items=[]):` preserve entries between calls?

- **A.** Lists are global by syntax
- **B.** The default list is created once when the definition executes
- **C.** append creates a new list
- **D.** Python caches every return value

3. How does a `None` default support fresh per-call list state?

- **A.** None behaves like a list
- **B.** It copies every caller object
- **C.** The body detects None and creates a new list during that call
- **D.** It prevents explicit list arguments

4. What does `*` mean before `precision` in `def f(value, *, precision=1):`?

- **A.** precision accepts any number of values
- **B.** value becomes optional
- **C.** All arguments become lists
- **D.** precision must be supplied by keyword if supplied

Write your choices before checking the answer key at the end of this book.

9. Read positional-only markers in documentation

Some built-ins display `/` in their signature. A small user-defined example is:

```python
def percentage(part, whole, /, *, precision=1):
    value = part / whole * 100
```

```
PYTHON (CONTINUED)
    return round(value, precision)


assert percentage(1, 4) == 25.0
assert percentage(1, 3, precision=2) == 33.33
```

Parameters before `/` are positional-only. `percentage(part=1, whole=4)` would raise `TypeError`. Parameters after `*` are keyword-only. The ordinary parameters between those markers, if any, may be supplied either way.

You do not need to make every interface positional-only. Learn to read the marker because it appears in Python's documentation and permits API designers to keep some parameter names from becoming caller commitments.

10. `*args` collects extra positional arguments

```
PYTHON
def total_distance(*distances):
    return sum(distances)


assert total_distance() == 0
assert total_distance(4, 7, 3) == 14
```

Inside the function, `distances` is a tuple. This interface is honest when any number of same-role values makes sense. It would be worse for a fixed record such as `(name, distance, unit)`, whose roles deserve explicit names.

Ordinary parameters can come first:

```
PYTHON
def route_message(name, *distances):
    return f"{name}: {sum(distances)} km"


assert route_message("ridge", 4, 7) == "ridge: 11 km"
```

11. `**kwargs` collects extra keyword arguments

```
PYTHON
def format_tags(**tags):
    parts = []
    for key, value in tags.items():
        parts.append(f"{key}={value}")
    return ", ".join(parts)


assert format_tags(zone="north", status="open") == "zone=north, status=open"
```

Inside, `tags` is a dictionary. Use this only when a genuinely open set of named options is part of the contract or when forwarding another interface. Do not replace three known parameters with `**kwargs`; that hides spelling errors and removes useful signature guidance.

12. Stars at a call unpack existing collections

A star in a call has a different role from a star in a definition:

```python
def coordinate_label(row, column):
    return f"({row}, {column})"


coordinate = (3, 7)
assert coordinate_label(*coordinate) == "(3, 7)"
```

`*coordinate` supplies its items as positional arguments.

Double-star supplies mapping entries as keyword arguments:

```python
settings = {"precision": 2, "unit": "miles"}
assert format_distance(12.345, **settings) == "12.35 miles"
```

Keys must match accepted keyword names. A duplicate still fails:

```python
# format_distance(12.345, precision=1, **settings)
```

Here both the explicit keyword and mapping try to bind `precision`.

13. Forward arguments without changing their call shape

A small forwarding wrapper can accept and pass through another function’s call:

```python
def call_and_label(function, *args, **kwargs):
    result = function(*args, **kwargs)
    return f"result={result}"


assert call_and_label(format_distance, 12.345, precision=2) == "result=12.35 km"
```

Lesson 8 uses this shape in decorators. Forwarding is a legitimate variadic use because the wrapper supports the wrapped callable’s argument interface rather than pretending its own domain has unknown fields.

Checkpoint: variadic and unpacked calls

Variadic and unpacking checkpoint

Distinguish stars in definitions from stars in calls.

1. Inside `def total(*values):`, what is `values`?

- **A.** A tuple of extra positional arguments
- **B.** A dictionary
- **C.** Only the final argument
- **D.** The function’s return value

2. Inside `def configure(**options):`, what is `options`?

- **A.** A tuple
- **B.** A dictionary of extra keyword arguments

- **C.** A set of parameter names
- **D.** A string

3. What does `coordinate_label(*(3, 7))` do?

- **A.** Passes the tuple as one argument
- **B.** Multiplies the tuple
- **C.** Supplies 3 and 7 as separate positional arguments
- **D.** Creates keyword arguments

4. What must be true for `format_distance(12, **settings)` to bind successfully?

- **A.** settings must be a list
- **B.** Every value must be text
- **C.** The function must return a dictionary
- **D.** The mapping keys must be accepted keyword names without duplicate bindings

Write your choices before checking the answer key at the end of this book.

14. Docstrings and annotations communicate; they do not coerce

```python
def repeat_message(message: str, times: int = 1) -> str:
    """Return message repeated times times."""
    return message * times


assert repeat_message("ha", 3) == "hahaha"
```

The docstring explains purpose. Annotations describe intended value roles to readers and tools. Python still permits a call such as `repeat_message("ha", "3")` to reach the body; annotations do not convert the string or automatically reject it. That operation then raises `TypeError`.

Keep the lesson boundary clear:

- Unit 5 uses annotations to make a signature legible.
- Unit 14 develops annotation design and static checking.
- Unit 15 develops docstring conventions and API documentation.

You can inspect an interface:

```python
from inspect import signature

assert str(signature(repeat_message)) == "(message: str, times: int = 1) -> str"
```

15. Build a mission-message interface

Complete this scaffold:

```python
def build_mission_message(
    name: str,
    distance: float,
    *,
    precision: int = 1,
    unit: str = "km",
    urgent: bool = False,
    notes=None,
) -> str:
    """Return one formatted mission message."""
    pass
```

Contract:

1. `name` and `distance` are the essential positional-or-keyword data.
2. All configuration is keyword-only.
3. `notes=None` creates fresh per-call notes storage; an explicitly supplied list is read but not changed.
4. Ignore empty note strings and join non-empty notes with `"; "` after the base message.
5. Prefix `"URGENT: "` only when requested.
6. Format distance with the requested precision and unit.
7. Return the string without printing.

Run:

```python
assert build_mission_message("Moon Pass", 12.345) == "Moon Pass: 12.3 km"
assert build_mission_message(
    "Moon Pass", 12.345, precision=2, unit="miles", urgent=True
) == "URGENT: Moon Pass: 12.35 miles"
assert build_mission_message(
    "Ridge", 7, notes=["windy", "", "bring rope"]
) == "Ridge: 7.0 km | windy; bring rope"
assert build_mission_message("Dock", 0, notes=[]) == "Dock: 0.0 km"

notes = ["night route"]
result = build_mission_message("Cave", 3, notes=notes)
assert notes == ["night route"]
assert result.endswith(" | night route")
```

Then call it with:

```python
mission_data = ("Garden", 5.5)
mission_options = {"precision": 0, "urgent": True}
```

Use both unpacking operators and prove the result is `"URGENT: Garden: 6 km"`.

Hint: separate binding from message assembly

Treat `notes is None` as no notes for this call. Build a list of non-empty note strings without changing the supplied collection. Assemble the base string, optional urgent prefix, and optional notes suffix in named stages.

Show one complete solution after attempting the lab

```python
def build_mission_message(
    name: str,
    distance: float,
    *,
    precision: int = 1,
    unit: str = "km",
    urgent: bool = False,
    notes=None,
) -> str:
    """Return one formatted mission message."""
    if notes is None:
        notes = []

    kept_notes = []
    for note in notes:
        if note:
            kept_notes.append(note)

    prefix = "URGENT: " if urgent else ""
    message = f"{prefix}{name}: {distance:.{precision}f} {unit}"
    if kept_notes:
        message += " | " + "; ".join(kept_notes)
    return message


mission_data = ("Garden", 5.5)
mission_options = {"precision": 0, "urgent": True}
unpacked_message = build_mission_message(*mission_data, **mission_options)
```

The function does not mutate an explicit notes list. The `None` sentinel still makes the absence of notes distinct from a shared mutable default.

16. Explain the interface

1. When are argument expressions evaluated relative to the body?
2. Which calls bind by order, and which bind by parameter name?
3. Why is a mutable default shared, and how does `None` change the timing?
4. How do `*` and `**` differ in definitions and calls?
5. What do annotations communicate without enforcing at runtime?

Key points

Key points

- Python evaluates arguments, binds parameters, and only then begins the body.
- Positional values bind by order; keyword values bind by name; duplicate, missing, extra, or unexpected bindings raise `TypeError`.
- Defaults are evaluated at definition time. Use a sentinel for fresh mutable per-call state.
- Keyword-only parameters make configuration visible; positional-only markers commonly appear in built-in documentation.
- `*args` and `**kwargs` collect variable arguments in definitions; `*iterable` and `**mapping` expand arguments at calls.

- Docstrings and annotations communicate the interface but do not perform runtime conversion or validation. :::

References

- Python tutorial: default argument values
- Python tutorial: keyword and special parameters
- Python tutorial: unpacking argument lists
- Python standard library: `inspect.signature`

Breaking a Program into Small Functions

Online lesson and runnable notebook

Split one useful program into cohesive calculations and presentation steps with explicit inputs, returned results, side effects, and empty-input policies.

- **Level:** Python Foundations · Unit 5
- **Estimated time:** 3.5–5 hours
- **You will learn:** Give each function one coherent responsibility, compose returned values, and keep calculation separate from deliberate printing or mutation.
- **Practice in:** Google Colab, JupyterLab, or a local editor

1. One long report is difficult to change safely

Imagine a trail game that records checkpoints reached by each player:

PYTHON

```
records = [
    {"player": "Ari", "distances": [3, 4, 2]},
    {"player": "Bo", "distances": [5, 1]},
    {"player": "Cy", "distances": []},
]

lines = []
for record in records:
    if not record["player"]:
        continue
    distance = sum(record["distances"])
    if distance >= 8:
        status = "pathfinder"
    elif distance > 0:
        status = "explorer"
    else:
        status = "ready"
    lines.append(f'{record["player"]}: {distance} km — {status}')

headline = f"Trail report: {len(lines)} players"
print(headline)
for line in lines:
    print(line)
```

The block works, but validation, calculation, classification, formatting, and display are tangled together. Changing the status boundary risks disturbing the format. Reusing the totals without printing means copying part of the block.

This lesson answers four practical questions:

- Where should one function end and another begin?
- Which values should cross each function boundary?
- How does returning data differ from printing or mutating it?
- How can small functions form one understandable program?

2. Describe the transformations before extracting functions

Start with observable examples, not arbitrary function names:

Input or situation	Required result
`[3, 4, 2]`	total distance 9
total 9	status `"pathfinder"`
name `"Ari"`, total 9, status `"pathfinder"`	one complete report line
empty distance list	total `0`, status `"ready"`
complete source records	unchanged after building the report

The table reveals a data path:

TEXT

```
distances → total → status ─┐
name ──────────────────────┼→ formatted line → displayed report
record count ──────────────┘
```

This is enough decomposition for now. Unit 7 develops a fuller method for ambiguous problems and algorithm choices. Here the problem is understood; the goal is to give its parts useful boundaries.

3. A contract says more than a function name

Before writing a body, record five facts:

Contract part	Question	Example
Purpose	What one useful job does it own?	Classify one distance total.
Inputs	What values must the caller supply?	A non-negative number.
Result	What value comes back?	A status string.
Side effects	What outside state changes?	None.
Boundary	What happens for zero?	Return `"ready"`.

The contract for `classify_distance` can appear as a docstring plus examples:

PYTHON

```
def classify_distance(distance):
    """Return the trail status for one non-negative distance total."""
    if distance >= 8:
        return "pathfinder"
    if distance > 0:
        return "explorer"
    return "ready"


assert classify_distance(9) == "pathfinder"
assert classify_distance(6) == "explorer"
assert classify_distance(0) == "ready"
```

The docstring states the supported domain. The assertions provide concrete acceptance evidence. Unit 13 will turn this idea into a systematic pytest test suite; these local assertions simply keep the lesson contract visible.

4. Calculation and presentation deserve different boundaries

Compare these two total functions:

```python
def show_total(distances):
    print(sum(distances))


def total_distance(distances):
    return sum(distances)
```

Both calculate. Only `total_distance` gives data back to its caller:

```python
distances = [3, 4, 2]
total = total_distance(distances)

assert total == 9
assert classify_distance(total) == "pathfinder"
```

`show_total` has a visible side effect but implicitly returns `None`. It cannot conveniently feed the next calculation. Returning a value keeps options open: the caller may compare it, store it, format it, or eventually print it.

Printing is not wrong. It belongs at the point where display is the intended effect:

```python
def display_report(report):
    """Print a prepared report and return None."""
    print(report)
```

The command-style name and docstring make that effect explicit.

5. Pure functions are easy to reuse, but effects still have a place

A **pure function** produces its result only from its inputs and does not change outside state:

```python
def add_bonus(score, bonus):
    return score + bonus


assert add_bonus(10, 2) == 12
assert add_bonus(10, 2) == 12
```

The same inputs give the same result. Contrast a mutation:

```python
def add_badge(badges, badge):
    """Append one badge to badges; return None."""
    badges.append(badge)
```

PYTHON (CONTINUED)

```
earned = ["starter"]
result = add_badge(earned, "pathfinder")

assert earned == ["starter", "pathfinder"]
assert result is None
```

Mutation is the stated job, so the contract names it. Unit 6 explains aliases, identity, and copies in detail. For now, inspect both the returned result and the supplied collection whenever a function may mutate.

Hidden global dependencies are harder to see:

PYTHON

```
STATUS_LIMIT = 8


def hidden_status(distance):
    return "pathfinder" if distance >= STATUS_LIMIT else "explorer"
```

A fixed module constant can be reasonable. A changing rule is clearer as an input:

PYTHON

```
def status_for(distance, pathfinder_at=8):
    return "pathfinder" if distance >= pathfinder_at else "explorer"


assert status_for(7) == "explorer"
assert status_for(7, pathfinder_at=6) == "pathfinder"
```

The caller can now see and control the dependency.

Checkpoint: responsibilities and contracts

Function responsibility checkpoint

Choose the boundary that makes each behavior explicit.

1. Which contract detail explains what `classify_distance(0)` should do?

- **A.** Its empty or boundary policy
- **B.** Its indentation width
- **C.** Its file name
- **D.** Its caller’s local variables

2. Why separate total calculation from line formatting?

- **A.** Python limits functions to one operator
- **B.** They have different inputs and different reasons to change
- **C.** Formatting cannot use return
- **D.** A function may not call another function

3. What should `add_badge` document because it appends to the supplied list?

- **A.** That every list becomes global
- **B.** That append returns the new list
- **C.** That it mutates the `badges` argument
- **D.** That it can accept no arguments

4. Which design makes a changing status threshold easiest for a caller to control?

- **A.** Read an undocumented global
- **B.** Repeat the number throughout the body
- **C.** Ask the user inside the calculation
- **D.** Accept a threshold parameter with a clear default

Write your choices before checking the answer key at the end of this book.

6. Queries return information; commands perform an effect

A **query** asks for information:

```python
def report_headline(player_count):
    return f"Trail report: {player_count} players"
```

A **command** asks the program to do something visible:

```python
def announce_headline(player_count):
    print(report_headline(player_count))
```

The distinction is a design aid, not an absolute law. A file-writing function may both perform an effect and return the number of bytes written. What matters is that its caller can discover both parts of the promise without reading every line of its body.

For beginner programs, calculate first and place the final effect near the outside boundary:

```python
headline = report_headline(3)
display_report(headline)
```

This ordering makes accidental repeated output less likely and lets you inspect the completed string before displaying it.

7. Compose helpers through visible intermediate values

Functions become useful when one returned result becomes another input:

```python
def build_player_line(record):
    distance = total_distance(record["distances"])
    status = classify_distance(distance)
    return f'{record["player"]}: {distance} km — {status}'


ari = {"player": "Ari", "distances": [3, 4, 2]}
```

PYTHON (CONTINUED)

```
line = build_player_line(ari)
assert line == "Ari: 9 km — pathfinder"
```

Trace the data, not just the output:

Step	Expression	Result kept by `build_player_line`
1	`record["distances"]`	`[3, 4, 2]`
2	`total_distance(...)`	9 in `distance`
3	`classify_distance(distance)`	`"pathfinder"` in `status`
4	formatted string	complete line returned to caller

This could be compressed into nested calls, but intermediate names are valuable when each result has meaning or may need inspection. Concision is not the same as clarity.

8. Pass only what a helper needs

Suppose a formatter receives all records even though it uses one name and two summary values. That broad input hides its true contract. Prefer:

PYTHON

```
def format_player_line(name, distance, status):
    return f"{name}: {distance} km — {status}"


assert format_player_line("Ari", 9, "pathfinder") == (
    "Ari: 9 km — pathfinder"
)
```

Now formatting can be checked without reconstructing an entire game. The caller owns the data flow:

PYTHON

```
distance = total_distance(ari["distances"])
status = classify_distance(distance)
line = format_player_line(ari["player"], distance, status)
```

Do not interpret this as “always use many scalar parameters.” A cohesive record can be the right input. The rule is to avoid making a function search unrelated state for the few values it actually needs.

9. Split by reasons to change, not by line count

A kitchen-sink function may validate, calculate, choose policy, format, and print. Those responsibilities change for different reasons. A report format change should not risk its distance calculation.

The opposite extreme also hurts:

PYTHON

```
def add(left, right):
    return left + right


def turn_into_text(value):
```

PYTHON (CONTINUED)

```
    return str(value)
```

If these wrappers merely rename obvious operators once, they lengthen the call chain without adding a meaningful contract. Extraction earns its place when it provides reuse, independent policy, a meaningful name, or an independently checkable transformation.

Use this decision list:

1. Can the job be named as one short promise?
2. Are its inputs and result narrower than the whole program?
3. Might it be reused or checked separately?
4. Does it change for a different reason from its neighbors?

One “yes” can be enough. A line-count limit is not required.

Checkpoint: returns and effects

Data flow and side effects checkpoint

Follow what crosses each function boundary.

1. What result is assigned by `saved = show_total([2, 3])` when `show_total` only prints?

- **A.** 5
- **B.** None
- **C.** [2, 3]
- **D.** The text `saved`

2. What is expected from two calls to the pure function `add_bonus(10, 2)`?

- **A.** The second call must fail
- **B.** The input becomes 12
- **C.** Both calls return 12 without outside state changes
- **D.** The second call returns 14

3. Which function is most clearly a command?

- **A.** One that returns a distance
- **B.** One that classifies a score
- **C.** One that formats a string
- **D.** One whose documented purpose is printing a prepared report

4. Why keep `distance` and `status` as intermediate names in a call chain?

- **A.** They expose meaningful stages that can be inspected and explained
- **B.** Python forbids nested calls
- **C.** They make both values global
- **D.** They cause every helper to print

Write your choices before checking the answer key at the end of this book.

10. A result can include both an answer and a reason

A plain status may not tell a caller why it was chosen. Return related evidence together when the caller needs both:

PYTHON
```
def classify_with_reason(distance):
    if distance >= 8:
        return {"status": "pathfinder", "reason": "distance reached 8 km"}
    if distance > 0:
        return {"status": "explorer", "reason": "distance is between 1 and 7 km"}
    return {"status": "ready", "reason": "no distance recorded"}


result = classify_with_reason(9)
assert result["status"] == "pathfinder"
assert result["reason"] == "distance reached 8 km"
```

The dictionary labels the fields. A two-item tuple can also be suitable when the positions are obvious. Avoid returning a long unexplained tuple that forces every caller to memorize positions.

11. Source preservation is part of the contract

The following formatter reads records without changing them:

PYTHON
```
def build_lines(records):
    lines = []
    for record in records:
        distance = total_distance(record["distances"])
        status = classify_distance(distance)
        line = format_player_line(record["player"], distance, status)
        lines.append(line)
    return lines
```

Record a small snapshot before the call:

PYTHON
```
records = [{"player": "Ari", "distances": [3, 4, 2]}]
snapshot = [{"player": "Ari", "distances": [3, 4, 2]}]

lines = build_lines(records)

assert records == snapshot
assert lines == ["Ari: 9 km — pathfinder"]
```

Equality is enough for this simple nested fixture. Unit 6 will explain why copying nested mutable data deserves more care.

12. Lab: assemble an expedition report

Build a small pipeline with these contracts:

PYTHON
```
def valid_record(record):
    """Return whether a record has a non-empty name and a distances list."""
    raise NotImplementedError
```

PYTHON (CONTINUED)

```
def summarize_record(record):
    """Return a summary dictionary for one supported record."""
    raise NotImplementedError


def format_summary(summary):
    """Return one display line for a summary dictionary."""
    raise NotImplementedError


def build_expedition_report(records):
    """Return a complete report string without changing records."""
    raise NotImplementedError
```

Use this source and keep it unchanged:

PYTHON

```
expedition = [
    {"name": "Ari", "distances": [3, 4, 2]},
    {"name": "Bo", "distances": [5, 1]},
    {"name": "Cy", "distances": []},
    {"name": "", "distances": [100]},
]

expected_report = """Expedition report — 3 travelers
Ari: 9 km — pathfinder
Bo: 6 km — explorer
Cy: 0 km — ready"""

assert valid_record(expedition[0]) is True
assert valid_record(expedition[-1]) is False
assert summarize_record(expedition[0]) == {
    "name": "Ari",
    "distance": 9,
    "status": "pathfinder",
}
assert format_summary({"name": "Cy", "distance": 0, "status": "ready"}) == (
    "Cy: 0 km — ready"
)

before = [
    {"name": "Ari", "distances": [3, 4, 2]},
    {"name": "Bo", "distances": [5, 1]},
    {"name": "Cy", "distances": []},
    {"name": "", "distances": [100]},
]
assert build_expedition_report(expedition) == expected_report
assert expedition == before
assert build_expedition_report([]) == "Expedition report — 0 travelers"
```

Work from the inner calculations outward. Do not print inside these four functions. After the assertions pass, display the returned report once.

Hint: make each stage return the next stage’s input

Filter with `valid_record`, transform each valid record with `summarize_record`, format each summary, and finally join the headline and lines with `"\n".join(...)`.

Show one complete solution after attempting the lab

PYTHON

```
def valid_record(record):
    """Return whether a record has a non-empty name and a distances list."""
    return bool(record.get("name")) and isinstance(record.get("distances"), list)


def summarize_record(record):
    """Return a summary dictionary for one supported record."""
    distance = sum(record["distances"])
    if distance >= 8:
        status = "pathfinder"
    elif distance > 0:
        status = "explorer"
    else:
        status = "ready"
    return {"name": record["name"], "distance": distance, "status": status}


def format_summary(summary):
    """Return one display line for a summary dictionary."""
    return (
        f'{summary["name"]}: {summary["distance"]} km — '
        f'{summary["status"]}'
    )


def build_expedition_report(records):
    """Return a complete report string without changing records."""
    summaries = []
    for record in records:
        if valid_record(record):
            summaries.append(summarize_record(record))

    headline = f"Expedition report — {len(summaries)} travelers"
    lines = [headline]
    for summary in summaries:
        lines.append(format_summary(summary))
    return "\n".join(lines)
```

Checkpoint: composition and preservation

Composed report checkpoint

Trace the report pipeline and its source data.

1. Which order matches the expedition report’s data flow?

- **A.** Print, mutate, then validate
- **B.** Format every raw value, then guess totals
- **C.** Validate, summarize, format, then join
- **D.** Join records before calling any helper

2. Where should a record with an empty name be excluded?

- **A.** Inside every formatter
- **B.** Only after the report is printed
- **C.** By changing the source list
- **D.** At the explicit `valid_record` boundary

3. What does `build_expedition_report([])` return?

- **A.** A headline stating that there are 0 travelers
- **B.** None after printing
- **C.** An exception because no lines exist
- **D.** The original list

4. Why compare `expedition` with `before` after the call?

- **A.** To force all helpers to be global
- **B.** To provide evidence that the pipeline preserved its source
- **C.** To sort the report
- **D.** To make print return a string

Write your choices before checking the answer key at the end of this book.

13. Explain the design aloud

Use the completed lab to answer:

1. Which helper is a query, and where would a display command belong?
2. What precise value crosses each call boundary?
3. Which function owns the status policy?
4. Why is the source-preservation assertion useful?
5. Which two helpers might change for different reasons?

Then modify the pathfinder threshold in one deliberate place. If you must edit several unrelated functions, the responsibility boundary needs another look.

Key points

Key points

- A useful function owns one coherent responsibility, not an arbitrary number of lines.
- Contracts state purpose, inputs, returned results, side effects, and important boundaries.
- Returned data composes naturally; printing and mutation are effects that should be explicit.
- Pure functions are especially easy to reuse, but command-style functions are appropriate when their documented job is an effect.
- Intermediate names make a function pipeline observable and easier to explain.
- Split a large function by reasons to change, while avoiding meaningless wrappers around every tiny expression. :::

References

- Python tutorial: defining functions
- Python tutorial: documentation strings
- Python built-in function: `sum`

Following Names Through Function Calls

Online lesson and runnable notebook

Trace separate call frames and return points, follow Python's name lookup rules, repair shadowing and assignment mistakes, and build small closures with independent state.

- **Level:** Python Foundations · Unit 5
- **Estimated time:** 3.5–5 hours
- **You will learn:** Follow local frames and return points, apply LEGB lookup, diagnose shadowing and `UnboundLocalError`, and use `nonlocal` state deliberately.
- **Practice in:** Google Colab, JupyterLab, or a local editor

1. The same name can be safe in separate calls

This dispatch program uses `message` in more than one function:

```python
def add_priority(message):
    message = f"PRIORITY: {message}"
    return message


def build_dispatch(destination, message):
    prepared = add_priority(message)
    return f"{destination} <- {prepared}"


report = build_dispatch("North Gate", "bridge closed")
assert report == "North Gate <- PRIORITY: bridge closed"
```

Changing `message` inside `add_priority` does not overwrite the caller’s local `message`. Each active call has its own frame: a workspace holding that call’s parameters and local names.

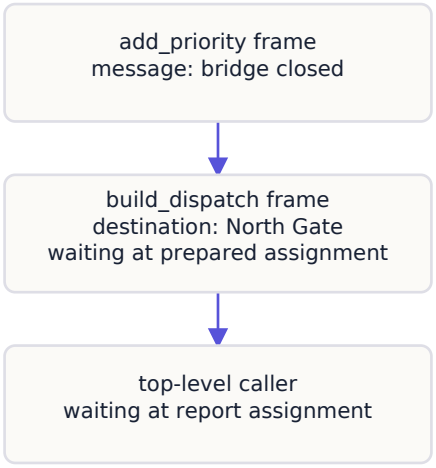
This lesson will help you answer:

- Which frame owns a name at a particular moment?
- Where does Python look when a name is not local?
- Why can an assignment cause `UnboundLocalError`?
- How can a returned function remember state from an enclosing call?

2. A call frame remembers where execution must return

When `build_dispatch` calls `add_priority`, the caller pauses. Python retains the caller frame and the location that should receive the returned value.

The active call is on top. Its return moves control and a value back to the suspended caller below it.



add_priority frame / message: bridge closed → build_dispatch frame / destination: North Gate / waiting at prepared assignment: returns prepared text

build_dispatch frame / destination: North Gate / waiting at prepared assignment → top-level caller / waiting at report assignment: returns report

Trace the active frames:

Moment	Active frame	Important local state	Waiting return point
1	`build_dispatch`	`destination`, `message`	top-level `report = ...`
2	`add_priority`	its own `message`	caller's `prepared = ...`
3	`build_dispatch`	new `prepared` value	top-level assignment
4	top level	new `report` value	none

After a function returns, its ordinary local frame is removed. A later call gets a fresh frame.

3. Repeated calls receive independent local state

PYTHON

```
def count_words(text):
    count = len(text.split())
    return count


first = count_words("red moon")
second = count_words("quiet silver river")

assert first == 2
assert second == 3
```

The first call's local `text` and `count` do not become inputs to the second. Identical parameter names describe roles within a call, not one shared storage box across all calls.

Recursion creates several frames for the same function at once. Lesson 6 will trace that special case. The rule remains the same: each active call owns its own parameters and locals.

4. Python looks for names through LEGB

When Python evaluates a name, it searches outward:

1. **Local** — the current function call;
2. **Enclosing** — any active enclosing function scope captured by a nested function;
3. **Global** — the module or notebook cell namespace;
4. **Built-in** — names such as `len`, `sum`, and `print`.

PYTHON
```
THEME = "moon"  # global


def make_label(prefix):  # prefix belongs to an enclosing function scope
    separator = ": "

    def label(name):  # name is local to this call
        return prefix + separator + name + " / " + THEME.upper()

    return label


warning_label = make_label("Warning")
assert warning_label("North Gate") == "Warning: North Gate / MOON"
```

Inside `label`, Python finds `name` locally, `prefix` and `separator` in the enclosing scope, `THEME` globally, and `str.upper` through the value's type. A separate built-in example is `len(name)`; `len` is found in built-ins when no closer scope defines it.

"Search outward" applies to reading a name. Assignment has an additional rule that we will make visible shortly.

5. Constants can be global; changing dependencies are clearer as inputs

A stable configuration constant is readable in uppercase:

PYTHON
```
MAX_LABEL_LENGTH = 20


def label_fits(text):
    return len(text) <= MAX_LABEL_LENGTH
```

A threshold that changes per dispatch is better passed explicitly:

PYTHON
```
def label_fits(text, max_length=20):
    return len(text) <= max_length


assert label_fits("North Gate") is True
assert label_fits("North Gate", max_length=5) is False
```

The second signature exposes the dependency and lets two callers use different rules without changing shared global state.

Checkpoint: call frames and lookup

Call frame checkpoint

Locate each name and return point before choosing.

1. Why does assigning `message` inside `add_priority` not replace `build_dispatch`'s parameter?

- **A.** Each active call has its own local frame
- **B.** Strings cannot be assigned
- **C.** All parameters are global
- **D.** Return blocks every assignment

2. What happens to `build_dispatch` while `add_priority` runs?

- **A.** Its frame is deleted
- **B.** Its frame waits at the call until a value returns
- **C.** It restarts from its first line
- **D.** Its locals move into built-ins

3. Where does nested `label` find `prefix`?

- **A.** Built-ins
- **B.** The module global namespace
- **C.** The enclosing `make_label` scope
- **D.** A previous unrelated call

4. How should a per-call maximum label length usually reach `label_fits`?

- **A.** By overwriting `len`
- **B.** Through an unrelated closure
- **C.** By editing the function body
- **D.** Through an explicit parameter

Write your choices before checking the answer key at the end of this book.

6. Shadowing hides an outer name

A local name can temporarily hide a global name:

```python
signal = "global signal"


def inspect_signal():
    signal = "local signal"
    return signal


assert inspect_signal() == "local signal"
assert signal == "global signal"
```

This is legal. It becomes confusing when the hidden name is needed. Shadowing a built-in is especially inconvenient:

```python
numbers = [2, 3, 5]
sum = 10

# total = sum(numbers)
```

If the commented line runs, Python tries to call integer `10`, not built-in `sum`. Rename the local value and remove the shadowing binding:

```python
del sum
recorded_total = 10
calculated_total = sum(numbers)

assert recorded_total == calculated_total
```

Other common names to protect include `list`, `str`, `min`, `max`, and `input`. The problem is not a special prohibition; it is ordinary nearest-scope lookup finding the wrong object first.

7. Assignment normally makes a name local to the whole function

This code looks as though it reads a global counter and then updates it:

```python
attempts = 3


def add_attempt():
    # attempts = attempts + 1
    return "uncomment the assignment to observe UnboundLocalError"
```

If `attempts = attempts + 1` is uncommented and the function is called, Python raises `UnboundLocalError`. Because the body assigns to `attempts`, Python treats that name as local throughout the function. The right side then tries to read the local value before that local value has been assigned.

The order of the lines does not turn the earlier read into a global read. Local scope is determined when the function is compiled.

Prefer explicit data flow:

```python
def increment_attempts(attempts):
    return attempts + 1


attempts = increment_attempts(attempts)
assert attempts == 4
```

The function can now be understood from its input and result.

8. `global` is real syntax, but it creates shared changing state

Python can explicitly rebind a global:

```python
signal_count = 0


def record_signal():
    global signal_count
    signal_count += 1


record_signal()
assert signal_count == 1
```

This works. It also means any call may change state used elsewhere, so the function's behavior depends on call history. A parameter and return value are usually easier for ordinary calculations. `global` is not forbidden; it is a strong coupling choice that should be visible in the contract.

9. A closure remembers an enclosing value

A nested function may survive after its enclosing call has returned:

```python
def make_channel_label(channel):
    prefix = f"Channel {channel}"

    def label(message):
        return f"{prefix}: {message}"

    return label


north_label = make_channel_label("North")
south_label = make_channel_label("South")

assert north_label("ready") == "Channel North: ready"
assert south_label("ready") == "Channel South: ready"
```

`label` is returned as a function object. Each returned function retains access to the enclosing `prefix` created by its own `make_channel_label` call. That retained relationship is a **closure**.

Definition and call times matter:

1. Calling `make_channel_label("North")` creates `prefix` and defines `label`.
2. Returning `label` does not call it.
3. Calling `north_label("ready")` later creates a new local frame for `message`.
4. Name lookup finds remembered `prefix` in the enclosing scope.

Checkpoint: shadowing and assignment

Name assignment checkpoint

Use scope rules rather than the intended English meaning of a name.

1. Why does `sum = 10` make a later `sum(numbers)` fail in that scope?

- **A.** Lists cannot be summed
- **B.** The nearest name `sum` now refers to an integer
- **C.** Built-ins work only once

- **D.** Numbers are enclosing functions

2. Why can `attempts = attempts + 1` raise `UnboundLocalError` inside a function?

- **A.** Addition is not allowed in functions
- **B.** Global integers cannot be read
- **C.** The assignment makes `attempts` local, so the right side reads it before local assignment
- **D.** Return must appear first

3. What important cost comes with rebinding a global counter?

- **A.** The function cannot have a name
- **B.** The counter becomes immutable
- **C.** The function automatically becomes recursive
- **D.** Behavior depends on shared state and call history

4. What does `north_label` remember after `make_channel_label` returns?

- **A.** The enclosing prefix created for North
- **B.** Every future global variable
- **C.** The result of calling `label`
- **D.** The caller's complete notebook

Write your choices before checking the answer key at the end of this book.

10. `nonlocal` updates remembered enclosing state

Without `nonlocal`, assigning to `count` would create a new local name in `record`. Declare that the assignment targets the enclosing binding:

PYTHON

```
def make_counter(label):
    count = 0

    def record():
        nonlocal count
        count += 1
        return f"{label}: {count}"

    return record


north_count = make_counter("North")
south_count = make_counter("South")

assert north_count() == "North: 1"
assert north_count() == "North: 2"
assert south_count() == "South: 1"
assert north_count() == "North: 3"
```

The two closures have independent state because two calls to `make_counter` created two enclosing `count` bindings. State persists between calls to one returned closure, but it is not one global counter shared by every instance.

Use closure state when it makes a small behavior cohesive. If callers need to inspect and coordinate many pieces of state, Unit 11's objects may offer a more explicit design.

11. Closures created in a loop can share the final loop value

Names in closures are looked up when the returned function runs, not copied automatically on every loop iteration:

```
PYTHON
def make_bad_labels():
    labels = []
    for channel in ["North", "South"]:
        labels.append(lambda: channel)
    return labels


bad_labels = make_bad_labels()
assert [label() for label in bad_labels] == ["South", "South"]
```

Both functions close over the same `channel` binding. After the loop, its value is `"South"`. A beginner-readable repair creates a new enclosing binding through a factory call:

```
PYTHON
def make_fixed_label(channel):
    def label():
        return channel

    return label


labels = []
for channel in ["North", "South"]:
    labels.append(make_fixed_label(channel))

assert [label() for label in labels] == ["North", "South"]
```

You may also encounter a default-argument capture, but the named factory makes the new binding and its timing easier to explain.

12. Lab: trace a dispatch center

Complete the pipeline and counter factory:

```
PYTHON
def normalize_message(message):
    """Return message stripped and converted to sentence case."""
    raise NotImplementedError


def route_message(channel, message, *, prefix="Dispatch"):
    """Return one routed message using explicit configuration."""
    raise NotImplementedError


def build_dispatch(channel, message, *, prefix="Dispatch"):
    """Normalize and route one message."""
    raise NotImplementedError


def make_dispatch_counter(channel):
    """Return a function that records and reports this channel's call count."""
```

PYTHON (CONTINUED)

```
    raise NotImplementedError
```

Use the assertions as call-frame evidence:

PYTHON

```
assert normalize_message("  BRIDGE CLOSED  ") == "Bridge closed"
assert route_message("North", "Bridge closed") == (
    "Dispatch North: Bridge closed"
)
assert build_dispatch("North", "  BRIDGE CLOSED  ") == (
    "Dispatch North: Bridge closed"
)
assert build_dispatch("South", "clear", prefix="Moon Base") == (
    "Moon Base South: Clear"
)

north = make_dispatch_counter("North")
south = make_dispatch_counter("South")
assert north() == "North dispatch 1"
assert north() == "North dispatch 2"
assert south() == "South dispatch 1"
assert north() == "North dispatch 3"
```

Before implementing, draw the stack while `build_dispatch` waits for `normalize_message`. After it returns, mark which frame owns `clean_message`.

Hint: distinguish the pipeline’s locals from remembered counter state

`build_dispatch` keeps `clean_message` only for that call and passes it to `route_message`. The factory creates `count` once; its nested function declares `nonlocal count` before incrementing it.

Show one complete solution after attempting the lab

PYTHON

```
def normalize_message(message):
    """Return message stripped and converted to sentence case."""
    return message.strip().capitalize()


def route_message(channel, message, *, prefix="Dispatch"):
    """Return one routed message using explicit configuration."""
    return f"{prefix} {channel}: {message}"


def build_dispatch(channel, message, *, prefix="Dispatch"):
    """Normalize and route one message."""
    clean_message = normalize_message(message)
    return route_message(channel, clean_message, prefix=prefix)


def make_dispatch_counter(channel):
    """Return a function that records and reports this channel's call count."""
    count = 0

    def record():
        nonlocal count
        count += 1
        return f"{channel} dispatch {count}"

    return record
```

Checkpoint: closures and independent state

Closure state checkpoint

Trace factory creation separately from calls to returned functions.

1. Which binding does `nonlocal count` update?

- **A.** A built-in count function
- **B.** The enclosing `make_dispatch_counter` call's `count`
- **C.** A new local created on each line
- **D.** Every global integer

2. Why does the first `south()` call report 1 after two `north()` calls?

- **A.** South calls ignore addition
- **B.** The label resets every integer
- **C.** Each factory call created its own enclosing `count`
- **D.** Both functions read a global zero

3. Why do both functions from `make_bad_labels` return `South`?

- **A.** Lambda sorts the strings
- **B.** North is not a valid value
- **C.** Returned functions always share all locals
- **D.** Both close over one loop binding whose final value is `South`

4. Where does `clean_message` live while `build_dispatch` calls `route_message`?

- **A.** In the suspended `build_dispatch` frame
- **B.** Inside built-ins
- **C.** In the earlier factory call
- **D.** In every frame at once

Write your choices before checking the answer key at the end of this book.

13. Explain the frames before moving on

Use your lab to answer:

1. At what line does `build_dispatch` pause during its helper call?
2. Which names are local to `route_message`?
3. Why is `prefix` explicit instead of read from changing global state?
4. When is each counter's enclosing `count` created?
5. What observable evidence proves the counters are independent?

Restart the notebook and call `north()` once. A clean restart creates the factory and its state again; it should report `1`, not remember an earlier run.

Key points

Key points

- Each active function call has a separate frame containing parameters, local names, and a return point.
- Python reads names through local, enclosing, global, and built-in scopes.
- Shadowing makes the nearest binding win, even when it hides a needed built-in.
- Assignment normally makes a name local throughout a function; reading it first can raise `UnboundLocalError`.
- `global` rebinds module state; explicit parameters and returns usually reveal changing dependencies more clearly.
- A closure retains access to enclosing bindings, and `nonlocal` can update one deliberately.
- Each factory call can create independent closure state. :::

References

- Python tutorial: scopes and namespaces
- Python language reference: naming and binding
- Python language reference: `global` and `nonlocal`

Passing Functions as Values

Online lesson and runnable notebook

Store, pass, and return functions deliberately; define callback contracts; and configure selection, transformation, and ranking behavior without duplicating traversal.

- **Level:** Python Foundations · Unit 5
- **Estimated time:** 3.5–5 hours
- **You will learn:** Distinguish a function object from a call result, pass behavior through callback contracts, return configured functions, and choose named functions or lambdas for readability.
- **Practice in:** Google Colab, JupyterLab, or a local editor

1. Keep one traversal and swap the policy

A tournament organizer wants several leaderboards from the same records:

```python
players = [
    {"name": "Ari", "score": 90, "solved": 5, "active": True},
    {"name": "Bo", "score": 75, "solved": 7, "active": False},
    {"name": "Cy", "score": 90, "solved": 6, "active": True},
]
```

Copying a loop for “active players,” “high scorers,” and “rank by solved puzzles” duplicates traversal. Instead, keep the loop stable and supply the part that varies as another function.

This requires precise answers to four questions:

- Is an expression referring to a function or calling it now?
- What value will the reusable code pass to a callback?
- What must the callback return?
- When is a named function clearer than a short `lambda`?

2. A function definition creates a value

```python
def by_score(player):
    return player["score"]


ranking_rule = by_score

assert callable(by_score) is True
assert callable(ranking_rule) is True
assert ranking_rule.__name__ == "by_score"
assert ranking_rule(players[0]) == 90
```

`by_score` without parentheses refers to the function object. Assignment binds the same object to another name; it does not run the body. Parentheses in `ranking_rule(players[0])` create a call and produce the integer `90`.

Keep the distinction visible:

Expression	Meaning	Value produced immediately
`by_score`	refer to behavior	function object
`by_score(players[0])`	run behavior now	`90`
`callable(by_score)`	ask whether object can be called	`True`
`by_score.__name__`	inspect a useful metadata attribute	`"by_score"`

3. Pass the function, not its early result

`sorted` accepts a `key` callback:

```python
ranked = sorted(players, key=by_score, reverse=True)
assert [player["name"] for player in ranked] == ["Ari", "Cy", "Bo"]
```

Python does not call `by_score` while evaluating `key=by_score`. `sorted` later calls it once for each player and uses each returned score as the comparison key.

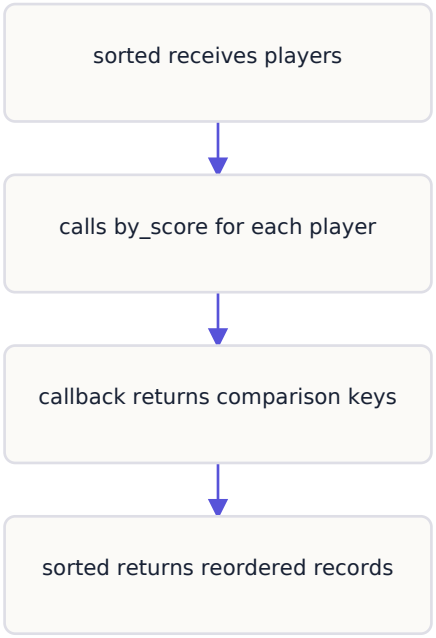
This is a different expression:

```python
early_result = by_score(players[0])
assert early_result == 90

# sorted(players, key=early_result)
```

If the commented call runs, `sorted` eventually tries to call integer `90` and raises `TypeError`. The early call replaced behavior with one result.

The sorting mechanism owns repetition; the callback owns the policy for one record.



4. A callback contract connects two functions

For `sorted(players, key=by_score)`, the contract has two sides:

Participant	Promise
`sorted`	Call key with one list item; use the returned key for ordering.
`by_score`	Accept one player record; return a comparable score.

The callback should not expect the complete list because `sorted` supplies one item. It should not return a dictionary if those dictionaries have no supported ordering. Read callback documentation as an interface between the mechanism and the supplied behavior.

Named key functions work with several built-ins:

```python
best = max(players, key=by_score)
lowest = min(players, key=by_score)

assert best["name"] == "Ari"
assert lowest["name"] == "Bo"
```

Ties retain stable input order in `sorted`. `max` returns the first maximum it encounters. Unit 3 introduced sorting behavior; here the new idea is how a callable supplies the key.

Checkpoint: function object or call result

Function value checkpoint

Decide whether each expression passes behavior or runs it immediately.

1. What does `ranking_rule = by_score` assign?

- **A.** The function object itself
- **B.** The score of every player
- **C.** A sorted list
- **D.** The text of the function body

2. Who calls `by_score` in `sorted(players, key=by_score)`?

- **A.** The assignment statement
- **B.** `sorted`, once for each item whose key it needs
- **C.** The dictionary constructor
- **D.** Nobody; functions cannot be arguments

3. Why is `key=by_score(players[0])` the wrong callback?

- **A.** The first player cannot be ranked
- **B.** Key functions must print
- **C.** It passes integer 90 rather than callable behavior
- **D.** Parentheses are forbidden in arguments

4. What argument should `by_score` expect when used as this key callback?

- **A.** No arguments
- **B.** The final ranked list
- **C.** Only a score integer
- **D.** One player record

Write your choices before checking the answer key at the end of this book.

5. Store actions behind stable command names

Functions can be values in collections:

```python
def cheer(name):
    return f"Go, {name}!"


def welcome(name):
    return f"Welcome, {name}."


commands = {
    "cheer": cheer,
    "welcome": welcome,
}

chosen = commands["cheer"]
assert chosen("Ari") == "Go, Ari!"
```

The dictionary stores a small, fixed set of known actions. The string chooses a function; a separate call supplies the function argument. Do not turn arbitrary user text into executable code. This pattern is useful because the allowed commands and their contracts remain explicit.

Every stored function here accepts one name and returns one string. A consistent contract lets the caller use the chosen function without a separate call shape for every key.

6. Write a reusable selector with a predicate callback

A **predicate** returns `True` or `False` for one item:

PYTHON
```
def is_active(player):
    return player["active"]


def has_high_score(player):
    return player["score"] >= 80
```

One explicit traversal can accept either policy:

PYTHON
```
def select_players(players, predicate):
    """Return players for which predicate(player) is true."""
    selected = []
    for player in players:
        if predicate(player):
            selected.append(player)
    return selected


active = select_players(players, is_active)
high_scores = select_players(players, has_high_score)

assert [player["name"] for player in active] == ["Ari", "Cy"]
assert [player["name"] for player in high_scores] == ["Ari", "Cy"]
```

`select_players` owns traversal. Each predicate owns one eligibility rule. The selector's docstring states exactly how it calls the callback and interprets the result.

For a one-off simple condition, a comprehension is direct:

PYTHON
```
active_names = [player["name"] for player in players if player["active"]]
assert active_names == ["Ari", "Cy"]
```

Callbacks become valuable when callers genuinely need to swap, reuse, or inject behavior—not as a requirement to replace every readable loop.

7. A transform callback returns a new representation

```python
def player_label(player):
    return f'{player["name"]} ({player["score"]})'


def transform_players(players, transform):
    transformed = []
    for player in players:
        transformed.append(transform(player))
    return transformed


assert transform_players(players, player_label) == [
    "Ari (90)",
    "Bo (75)",
    "Cy (90)",
]
```

The callback contract differs from the predicate contract. A predicate's result decides keep/skip. A transform's result becomes one output value. Generic names such as `callback` are not enough by themselves; document what its result means.

Checkpoint: callback contracts

Callback contract checkpoint

Match the supplied behavior to what the reusable mechanism expects.

1. What should a predicate passed to `select_players` return for one player?

- **A.** A complete sorted list
- **B.** A truth value deciding whether to keep that player
- **C.** A function definition string
- **D.** A changed global list

2. How does `transform_players` use `transform(player)`?

- **A.** As a loop condition only
- **B.** As a new callback
- **C.** As the next value appended to its result
- **D.** As the player's source index

3. What makes the `commands` dictionary easy for one caller to use?

- **A.** Every function changes global state
- **B.** Dictionary keys are Python code
- **C.** The actions are called at definition time
- **D.** All stored actions share a compatible input/result contract

4. When is a direct comprehension often clearer than introducing a callback processor?

- **A.** When the condition is short, local, and not meant to be swapped

- **B.** Whenever data has more than one item
- **C.** Only when functions are unavailable
- **D.** When arbitrary code must be executed

Write your choices before checking the answer key at the end of this book.

8. Return a configured function from a factory

Lesson 4 introduced closures. A factory can return a ranking policy that remembers configuration:

```python
def score_with_bonus(bonus_for_solved):
    def ranking_key(player):
        return player["score"] + player["solved"] * bonus_for_solved

    return ranking_key


plain_score = score_with_bonus(0)
puzzle_weighted = score_with_bonus(5)

assert plain_score(players[1]) == 75
assert puzzle_weighted(players[1]) == 110
```

The outer call chooses a policy setting once. The returned function applies that setting to many records:

```python
ranked = sorted(players, key=puzzle_weighted, reverse=True)
assert [player["name"] for player in ranked] == ["Cy", "Bo", "Ari"]
```

This is useful when several functions need separate configurations. A normal parameter is simpler when only one direct call needs the setting.

9. Use `lambda` for one short readable expression

A lambda expression creates an anonymous function:

```python
by_name = lambda player: player["name"]
assert by_name(players[0]) == "Ari"

alphabetical = sorted(players, key=lambda player: player["name"])
assert [player["name"] for player in alphabetical] == ["Ari", "Bo", "Cy"]
```

The expression after the colon is returned automatically. A lambda cannot contain statements such as an ordinary `for`, `if` statement, assignment, or `return` statement. Conditional expressions are possible, but a complicated lambda is harder to explain than a named function.

Prefer `def` when behavior:

- needs a descriptive reusable name;
- takes several steps;
- benefits from a docstring or annotations;
- needs separate assertions; or

- is subtle enough to deserve explanation.

10. Read `map` and `filter` without treating them as mandatory

You may encounter callback-oriented built-ins:

```python
names_from_map = list(map(lambda player: player["name"], players))
active_from_filter = list(filter(is_active, players))

assert names_from_map == ["Ari", "Bo", "Cy"]
assert [player["name"] for player in active_from_filter] == ["Ari", "Cy"]
```

In Python 3, `map` and `filter` return lazy iterators. Lesson 7 explains their consumption model. Comprehensions often state simple transformations and conditions more directly:

```python
names = [player["name"] for player in players]
active = [player for player in players if is_active(player)]
```

Choose the form that communicates the operation to your readers. Function values are a tool for configurable behavior, not a style contest.

11. Callback failures reveal a mismatched contract

Three mistakes deserve separate diagnoses:

1. `predicate(player)` was called before it was passed, so the selector receives one Boolean instead of a callable.
2. A ranking callback returns values that cannot be compared consistently.
3. A callback expects two parameters, but the mechanism supplies one item.

Do not hide these failures inside a broad `try`/`except`. Read which operation tried to call or compare which value. Unit 8 develops systematic error handling; here repair the mismatch between the two sides of the callback contract.

12. Lab: build a configurable leaderboard

Keep the source records unchanged while swapping eligibility and ranking:

```python
def active_player(player):
    """Return whether player is active."""
    raise NotImplementedError


def solved_at_least(minimum):
    """Return a predicate requiring at least minimum solved puzzles."""
    raise NotImplementedError


def leaderboard(players, eligible, rank_key):
    """Return eligible player names ordered by descending rank key."""
    raise NotImplementedError
```

Use these checks:

```python
players = [
    {"name": "Ari", "score": 90, "solved": 5, "active": True},
    {"name": "Bo", "score": 75, "solved": 7, "active": False},
    {"name": "Cy", "score": 90, "solved": 6, "active": True},
    {"name": "Di", "score": 80, "solved": 8, "active": True},
]
snapshot = [record.copy() for record in players]

def by_score(player):
    return player["score"]


def by_solved_then_score(player):
    return player["solved"], player["score"]


assert leaderboard(players, active_player, by_score) == ["Ari", "Cy", "Di"]
at_least_six = solved_at_least(6)
assert leaderboard(players, at_least_six, by_solved_then_score) == [
    "Di",
    "Bo",
    "Cy",
]
assert leaderboard(players, lambda player: True, by_score) == [
    "Ari",
    "Cy",
    "Di",
    "Bo",
]
assert players == snapshot
assert callable(at_least_six)
```

The `eligible` callback must accept one record and return a truth value. The `rank_key` callback must accept one record and return a comparable key. Stable sorting keeps Ari before Cy when their scores tie.

Hint: pass the callbacks onward without calling them too early

Create a list containing records for which `eligible(player)` is true. Then call `sorted(..., key=rank_key, reverse=True)` and extract names from that returned list. The predicate factory's nested function compares `player["solved"]` with the remembered minimum.

Show one complete solution after attempting the lab

```python
def active_player(player):
    """Return whether player is active."""
    return player["active"]


def solved_at_least(minimum):
    """Return a predicate requiring at least minimum solved puzzles."""
    def eligible(player):
        return player["solved"] >= minimum

    return eligible


def leaderboard(players, eligible, rank_key):
    """Return eligible player names ordered by descending rank key."""
    selected = []
    for player in players:
        if eligible(player):
            selected.append(player)
```

PYTHON (CONTINUED)

```
    ranked = sorted(selected, key=rank_key, reverse=True)
    return [player["name"] for player in ranked]
```

Checkpoint: factories and readable policies

Configurable behavior checkpoint

Choose the design that preserves a clear callback contract.

1. What does `solved_at_least(6)` return?

- **A.** A function that remembers minimum 6
- **B.** Every matching player immediately
- **C.** The integer 6
- **D.** A sorted list of names

2. Which use is a good fit for a lambda?

- **A.** A five-step validation algorithm
- **B.** A one-off key expression such as `lambda player: player['name']`
- **C.** A public function needing documentation
- **D.** Behavior with several statements

3. What does `by_solved_then_score` return for ranking?

- **A.** A changed source record
- **B.** A callback factory
- **C.** A comparable tuple containing solved count then score
- **D.** A printed leaderboard

4. Why does Ari remain before Cy when both have score 90 in the score-only leaderboard?

- **A.** Ari’s name is always forced first
- **B.** Callbacks reorder ties randomly
- **C.** Dictionaries compare their insertion order
- **D.** Python sorting is stable and preserves their source order for equal keys

Write your choices before checking the answer key at the end of this book.

13. Explain who owns each decision

Using the lab, identify:

1. which function owns traversal;
2. which callback owns eligibility;
3. which callback owns ranking;
4. when the factory runs and when its returned predicate runs; and

5 why source order still matters for ties.

Then add `by_name_length` as a named key. Do not change `leaderboard`. If the new policy works by passing a different function, the mechanism and callback contracts are properly separated.

Key points

Key points

- A function is a value; parentheses distinguish referring to it from calling it.
- A callback contract states what the mechanism supplies and what the callback must return.
- Functions can be assigned, stored in collections, passed to other functions, and returned from factories.
- Predicates decide keep/skip; transforms produce output values; key functions produce comparison values.
- Closures can remember callback configuration chosen by a factory call.
- Use a lambda for one short, local expression and a named function when behavior needs explanation, reuse, or several steps.
- Prefer a direct loop or comprehension when configurable behavior would only add ceremony. :::

References

- Python data model: callable objects
- Python built-in function: `sorted`
- Python built-in functions: `map` and `filter`
- Python tutorial: lambda expressions

Solving Smaller Problems with Recursion

Online lesson and runnable notebook

Recognize recursively shaped data, write reachable base and progress cases, trace calls and returning results, and choose a loop when the problem is naturally flat.

- **Level:** Python Foundations · Unit 5
- **Estimated time:** 3.5–5 hours
- **You will learn:** Match recursion to nested data, define base and smaller-problem cases, trace frame creation and return unwinding, and compare recursion with iteration.
- **Practice in:** Google Colab, JupyterLab, or a local editor

1. Nested quests contain smaller quests

A flat list has one level. A quest tree can contain branches that contain more branches:

PYTHON

```
quest = {
    "name": "Moon Gate",
    "reward": 1,
    "children": [
        {"name": "River Key", "reward": 3, "children": []},
        {
            "name": "Old Tower",
            "reward": 2,
            "children": [
                {"name": "Star Lens", "reward": 5, "children": []},
            ],
        },
    ],
}
```

Every node has the same shape: a name, a reward, and a list of child nodes. A child is itself a quest tree. That recursive data definition suggests a recursive function: solve one node, then ask the same function to solve each smaller child.

This lesson answers:

- What stops a recursive chain of calls?
- How does each call make the remaining problem smaller?
- Why do returned results move in the opposite direction from calls?
- When is an ordinary loop the clearer tool?

2. Write the stopping case before the recursive case

A recursive function needs two connected promises:

1. **Base case:** solve a smallest input without another recursive call.
2. **Recursive case:** reduce a larger input to one or more smaller inputs and combine their results.

For a countdown:

```python
def countdown(number):
    if number <= 0:          # base case
        return ["launch"]
    rest = countdown(number - 1)  # smaller problem
    return [number] + rest


assert countdown(3) == [3, 2, 1, "launch"]
assert countdown(0) == ["launch"]
```

`number <= 0` is reachable because every recursive call subtracts one. The function does not wait for a depth error to stop it; the contract deliberately defines a smallest supported case.

Countdown is useful for tracing, but a loop is simpler for real flat countdowns. Nested trees provide the stronger motivation later in the lesson.

3. Calls move down; results return up

In `countdown(3)`, the first call cannot finish `[3] + rest` until `countdown(2)` produces `rest`. Each caller waits with a partly completed expression.

Calls descend toward the base case. Finished values then return toward the original caller.



countdown 3 waits → countdown 2 waits: call with 2

countdown 2 waits → countdown 1 waits: call with 1

countdown 1 waits → base returns launch list: call with 0

base returns launch list → build 1 then launch: return upward

build 1 then launch → build 2, 1, then launch: return upward

build 2, 1, then launch → build 3, 2, 1, then launch: return upward

The frames are separate:

Active call	Local `number`	Suspended work	Child result arrives as
`countdown(3)`	3	`[3] + rest`	`[2, 1, "launch"]`
`countdown(2)`	2	`[2] + rest`	`[1, "launch"]`
`countdown(1)`	1	`[1] + rest`	`["launch"]`

Active call	Local `number`	Suspended work	Child result arrives as
`countdown(0)`	0	none; base case	returns `["launch"]`

Only after the deepest result exists can the waiting calls finish in reverse order.

4. A numeric return trace exposes suspended expressions

```python
def recursive_sum(number):
    if number == 0:
        return 0
    return number + recursive_sum(number - 1)


assert recursive_sum(4) == 10
```

Expand calls without skipping the return direction:

```text
recursive_sum(4)
4 + recursive_sum(3)
4 + (3 + recursive_sum(2))
4 + (3 + (2 + recursive_sum(1)))
4 + (3 + (2 + (1 + recursive_sum(0))))
4 + (3 + (2 + (1 + 0)))
10
```

Each local `number` remains in its own waiting frame. There is no global accumulator. The final value is assembled from returned child values.

Checkpoint: base cases and progress

Recursive progress checkpoint

Identify the stopping rule and the strictly smaller problem.

1. What is the base case in `countdown`?

- **A.** When `number <= 0`, return the launch list without another call
- **B.** Whenever a list has three items
- **C.** After Python raises an error
- **D.** When the caller prints

2. Why can `countdown(number - 1)` eventually reach the base case for a positive integer?

- **A.** Lists stop recursion automatically
- **B.** Each call reduces `number` by one
- **C.** Every return becomes global
- **D.** Python guesses the final value

3. What is `recursive_sum(3)` waiting to receive from `recursive_sum(2)`?

- **A.** A new function definition
- **B.** The global number 3

- **C.** The returned sum for the smaller problem
- **D.** A printed traceback

4. In what order do finished results travel after the base case?

- **A.** Directly to every frame at once
- **B.** Only toward deeper calls
- **C.** In random frame order
- **D.** Back through waiting callers in reverse call order

Write your choices before checking the answer key at the end of this book.

5. A leaf is the base case in a tree

A leaf node has no children. Counting leaves follows the shape directly:

```python
def count_leaves(node):
    if node["children"] == []:
        return 1

    total = 0
    for child in node["children"]:
        total += count_leaves(child)
    return total


assert count_leaves(quest) == 2
```

Line by line:

1 A leaf returns `1` without another call.

2 A branch creates local `total = 0` for that call.

3 Each child is a smaller tree with the same structure.

4 Each child count returns to the waiting parent frame.

5 The parent combines child results and returns its subtotal.

No child call edits a hidden shared counter. Separate returned subtotals make the flow observable and keep repeated calls independent.

6. Include the current node when totaling rewards

```python
def total_reward(node):
    total = node["reward"]
    for child in node["children"]:
        total += total_reward(child)
    return total


assert total_reward(quest) == 11
```

The current node contributes before the recursive loop. A leaf naturally works: its child loop runs zero times, then its own reward returns. This design does not need a separate leaf branch because the empty loop already supplies the base behavior.

Both styles are valid:

- `count_leaves` has an explicit leaf base case because a leaf contributes the distinct value `1`;
- `total_reward` uses an empty child list as an implicit no-more-calls boundary and always returns the current reward plus child totals.

7. Return paths instead of relying on global history

Finding a target must distinguish “found here,” “found below,” and “not found”:

```python
def find_quest_path(node, target):
    if node["name"] == target:
        return [node["name"]]

    for child in node["children"]:
        child_path = find_quest_path(child, target)
        if child_path is not None:
            return [node["name"]] + child_path

    return None


assert find_quest_path(quest, "Star Lens") == [
    "Moon Gate",
    "Old Tower",
    "Star Lens",
]
assert find_quest_path(quest, "Missing") is None
```

When a child finds the target, its path returns upward. Each parent prepends its own name. If all children return `None`, this call also returns `None`. Early return stops searching sibling branches after a match.

Checkpoint: combining nested results

Nested tree checkpoint

Trace one node and the results returned by its children.

1. What does `count_leaves` return for a node whose children list is empty?

- **A.** The node name
- **B.** 1
- **C.** None
- **D.** The complete tree

2. How does a branch compute its leaf total?

- **A.** By changing one hidden global
- **B.** By counting only its own name
- **C.** By adding the values returned for its child trees

- **D.** By printing every leaf

3. When does `find_quest_path` return `None` for a branch?

- **A.** As soon as it has children
- **B.** Only after a global error
- **C.** Whenever reward is zero
- **D.** After neither the branch nor any child contains the target

4. How does `Moon Gate` appear at the front of the returned path to `Star Lens`?

- **A.** Its waiting frame prepends its name to the successful child path
- **B.** The leaf edits every parent dictionary
- **C.** Python sorts node names
- **D.** A global path is reused

Write your choices before checking the answer key at the end of this book.

8. Empty, one-child, and deep branches test the contract

Small shapes reveal missing cases:

```python
quiet_room = {"name": "Quiet Room", "reward": 0, "children": []}
one_path = {
    "name": "Gate",
    "reward": 1,
    "children": [quiet_room],
}

assert count_leaves(quiet_room) == 1
assert total_reward(quiet_room) == 0
assert count_leaves(one_path) == 1
assert total_reward(one_path) == 1
assert find_quest_path(one_path, "Quiet Room") == ["Gate", "Quiet Room"]
```

An empty **children collection** is different from an absent root node. These functions require one valid node as their input. If the application permits `None` as “no tree,” that additional boundary must be stated and handled explicitly rather than guessed inside every recursive call.

9. Two recursion failures have different causes

No reachable base case:

```python
def broken_countdown(number):
    # return broken_countdown(number - 1)
    return "The commented call would never test a stopping condition."
```

No progress toward the base case:

```python
def unchanged_countdown(number):
    if number <= 0:
        return "launch"
```

PYTHON (CONTINUED)

```
    # return unchanged_countdown(number)
    return "The commented call repeats the same positive problem."
```

If the recursive lines are used, calls continue until Python raises `RecursionError` after too many nested frames. The error is a safety symptom, not a stopping strategy. Repair the algorithm by proving that every supported path reaches a base case through smaller work.

Also check that branch recursion actually receives `child`, not the unchanged parent `node`.

10. Prefer a loop when the data is naturally flat

The recursive countdown can be written directly:

PYTHON

```
def countdown_with_loop(number):
    values = []
    while number > 0:
        values.append(number)
        number -= 1
    values.append("launch")
    return values


assert countdown_with_loop(3) == [3, 2, 1, "launch"]
```

The loop uses one function frame and matches the flat sequence. Python does not automatically replace tail recursion with a loop, and recursion depth is limited. Choose recursion when the problem or data is recursively shaped and the base/ smaller-case explanation is clearer. Choose iteration for straightforward flat repetition.

11. A recursive generator can yield leaves later

Lesson 7 develops generators fully. For now, notice how `yield from` can delegate to each smaller tree:

PYTHON

```
def leaf_names(node):
    if not node["children"]:
        yield node["name"]
        return

    for child in node["children"]:
        yield from leaf_names(child)


assert list(leaf_names(quest)) == ["River Key", "Star Lens"]
```

The recursive shape is the same as `count_leaves`; instead of returning a total, the function produces leaf names one at a time. The next lesson explains when the body starts, where it pauses, and what exhaustion means.

12. Lab: map a nested quest tree

Implement three focused recursive functions:

```python
def count_leaves(node):
    """Return the number of nodes that have no children."""
    raise NotImplementedError


def total_reward(node):
    """Return this node's reward plus every descendant reward."""
    raise NotImplementedError


def find_quest_path(node, target):
    """Return root-to-target names, or None when target is absent."""
    raise NotImplementedError
```

Use this richer tree and preserve it:

```python
quest = {
    "name": "Moon Gate",
    "reward": 1,
    "children": [
        {"name": "River Key", "reward": 3, "children": []},
        {
            "name": "Old Tower",
            "reward": 2,
            "children": [
                {"name": "Star Lens", "reward": 5, "children": []},
                {"name": "Empty Loft", "reward": 0, "children": []},
            ],
        },
    ],
}

import copy

before = copy.deepcopy(quest)
assert count_leaves(quest) == 3
assert total_reward(quest) == 11
assert find_quest_path(quest, "Moon Gate") == ["Moon Gate"]
assert find_quest_path(quest, "Star Lens") == [
    "Moon Gate",
    "Old Tower",
    "Star Lens",
]
assert find_quest_path(quest, "Empty Loft") == [
    "Moon Gate",
    "Old Tower",
    "Empty Loft",
]
assert find_quest_path(quest, "Missing") is None
assert quest == before
```

Write the base behavior in words first. During one trace, record the node name, the active local subtotal or child path, and the value returned to the parent.

Hint: let each child return a complete result for its own subtree

For totals, begin with this node’s contribution and add the result of each child call. For path search, return immediately when the current name matches; otherwise ask each child and prepend the current name only to a non-`None` path.

Show one complete solution after attempting the lab

```python
def count_leaves(node):
    """Return the number of nodes that have no children."""
    if not node["children"]:
        return 1

    total = 0
    for child in node["children"]:
        total += count_leaves(child)
    return total


def total_reward(node):
    """Return this node's reward plus every descendant reward."""
    total = node["reward"]
    for child in node["children"]:
        total += total_reward(child)
    return total


def find_quest_path(node, target):
    """Return root-to-target names, or None when target is absent."""
    if node["name"] == target:
        return [node["name"]]

    for child in node["children"]:
        child_path = find_quest_path(child, target)
        if child_path is not None:
            return [node["name"]] + child_path
    return None
```

Checkpoint: recursion or iteration

Recursive fit checkpoint

Choose the form that matches the problem shape and stopping evidence.

1. Which implementation is usually clearer for an ordinary flat countdown in Python?

- **A.** A loop with an explicit changing counter
- **B.** Recursion with no base case
- **C.** A global list changed forever
- **D.** Repeatedly calling the same unchanged input

2. Why is recursion a natural fit for the quest tree?

- **A.** Dictionaries force recursion
- **B.** Each child is a smaller value with the same tree shape
- **C.** Loops cannot read names
- **D.** Rewards are always integers

3. What does `RecursionError` usually reveal in these examples?

- **A.** The correct base result
- **B.** Successful source preservation
- **C.** Too many nested calls, often because stopping or progress is wrong
- **D.** Automatic conversion to iteration

4. What does `yield from leaf_names(child)` delegate?

- **A.** Global name lookup
- **B.** Dictionary mutation
- **C.** Sorting policy
- **D.** Production of values from the child’s recursive generator

Write your choices before checking the answer key at the end of this book.

13. Explain both directions of the computation

Use the lab to tell two stories:

1. **Downward:** which smaller child reaches each new frame, and why will a base case eventually occur?
2. **Upward:** what exact subtotal or path returns to the parent, and how does the parent combine it?

Then add a new three-level branch and predict all three results before running. If you can only explain the final number, expand a small call tree again.

Key points

Key points

- Recursion fits a problem that contains smaller instances of the same problem.
- A base case returns without another recursive call; a recursive case must make measurable progress toward it.
- Every call has a separate frame and may wait for a smaller call’s result.
- Calls move toward the base case; combined results return through callers in the opposite direction.
- Return child subtotals or paths instead of relying on a hidden global accumulator.
- Check empty and leaf shapes, one-child branches, absent targets, and deeper nesting.
- Prefer iteration when the work is naturally flat and a loop states it more directly. :::

References

- Python tutorial: defining functions
- Python language reference: function calls
- Python exception reference: `RecursionError`

Producing Values One at a Time

Online lesson and runnable notebook

Trace iterable and iterator state, build finite generators that pause and resume, and compose lazy pipelines whose consumers can stop early.

- **Level:** Python Foundations · Unit 5
- **Estimated time:** 4–5.5 hours
- **You will learn:** Distinguish reusable iterables from stateful iterators, predict partial consumption and exhaustion, and write finite generator functions, expressions, delegation, and lazy pipelines.
- **Practice in:** Google Colab, JupyterLab, or a local editor

1. Sometimes the caller needs only the next useful reading

Suppose a remote sensor collected many values, but an alert needs only the first three readings above a threshold. Building every transformed result first can do unnecessary work. An iterator lets the consumer ask for one value at a time and stop early.

This lesson separates several related terms:

- an **iterable** can provide an iterator;
- an **iterator** remembers its current consumption position;
- a **generator function** defines paused production with `yield`;
- a **generator object** is the stateful iterator returned by calling that function; and
- a **generator expression** creates a compact lazy transformation.

The important question is not merely “is this lazy?” It is: which object owns the current position, when does code run, and what happens after exhaustion?

2. A list is iterable; `iter` creates an iterator

PYTHON

```
readings = [12, 18, 25]
cursor = iter(readings)

assert iter(cursor) is cursor
assert next(cursor) == 12
assert next(cursor) == 18
assert next(cursor) == 25
```

The list stores all three values and can create iterators. `cursor` is one iterator with a current position. Each `next(cursor)` returns the next item and advances that position.

A `for` loop performs this protocol for you: it gets an iterator, requests next values, and stops normally when the iterator signals that no value remains.

3. Exhaustion is persistent

After the third item, another plain `next(cursor)` raises `StopIteration`. Keep that failing call commented in the normal lesson run:

```
PYTHON
# next(cursor)
```

`next` also accepts a default result:

```
PYTHON
assert next(cursor, "no reading") == "no reading"
assert next(cursor, "still empty") == "still empty"
```

Exhaustion does not rewind the iterator. Once consumed, it stays consumed. A `for` loop catches the stop signal internally; ordinary application code rarely calls `StopIteration` handling directly.

4. Collections restart; iterator objects continue

Two loops over the list each start from a newly created iterator:

```
PYTHON
readings = [12, 18, 25]
assert [value for value in readings] == [12, 18, 25]
assert [value for value in readings] == [12, 18, 25]
```

Two loops over the same iterator share its progress:

```
PYTHON
cursor = iter(readings)
first = next(cursor)
remaining = [value for value in cursor]
after_exhaustion = [value for value in cursor]

assert first == 12
assert remaining == [18, 25]
assert after_exhaustion == []
```

Create independent positions by calling `iter` on the collection twice:

```
PYTHON
left = iter(readings)
right = iter(readings)

assert next(left) == 12
assert next(left) == 18
assert next(right) == 12
```

The list is shared source data, but each iterator remembers its own position.

Checkpoint: iterator state

Iterator state checkpoint

Track the position owned by each iterator object.

1. What does `iter(readings)` create from the list?

- **A.** An iterator with its own current position
- **B.** A permanently changed list
- **C.** The final sum
- **D.** A function definition

2. What changes after `next(cursor)` returns a value?

- **A.** The source list loses that value
- **B.** The iterator advances its remembered position
- **C.** Every iterator advances
- **D.** The caller becomes a generator

3. What does `next(cursor, 'empty')` return after cursor is exhausted?

- **A.** The first item again
- **B.** The cursor object
- **C.** The supplied string `empty`
- **D.** A new list

4. Why can `right` still return 12 after `left` has returned 12 and 18?

- **A.** Integers copy the loop
- **B.** `next` runs randomly
- **C.** The list was rebuilt automatically
- **D.** `left` and `right` are separate iterators with separate positions

Write your choices before checking the answer key at the end of this book.

5. Calling a generator function does not run its body yet

A function containing `yield` is a generator function:

```python
def countdown(start):
    print("generator body started")
    while start > 0:
        yield start
        start -= 1


numbers = countdown(3)
```

The assignment creates a generator object. It prints nothing yet. The first request begins execution:

```python
assert next(numbers) == 3
```

At that point Python:

1. creates local `start = 3` for this generator call;
2. prints the start message;

3 reaches `yield start`;

4 produces `3`; and

5 suspends the frame before `start -= 1`.

The call and first execution are separate moments. This timing is one of the most important differences between an ordinary function call and a generator function call.

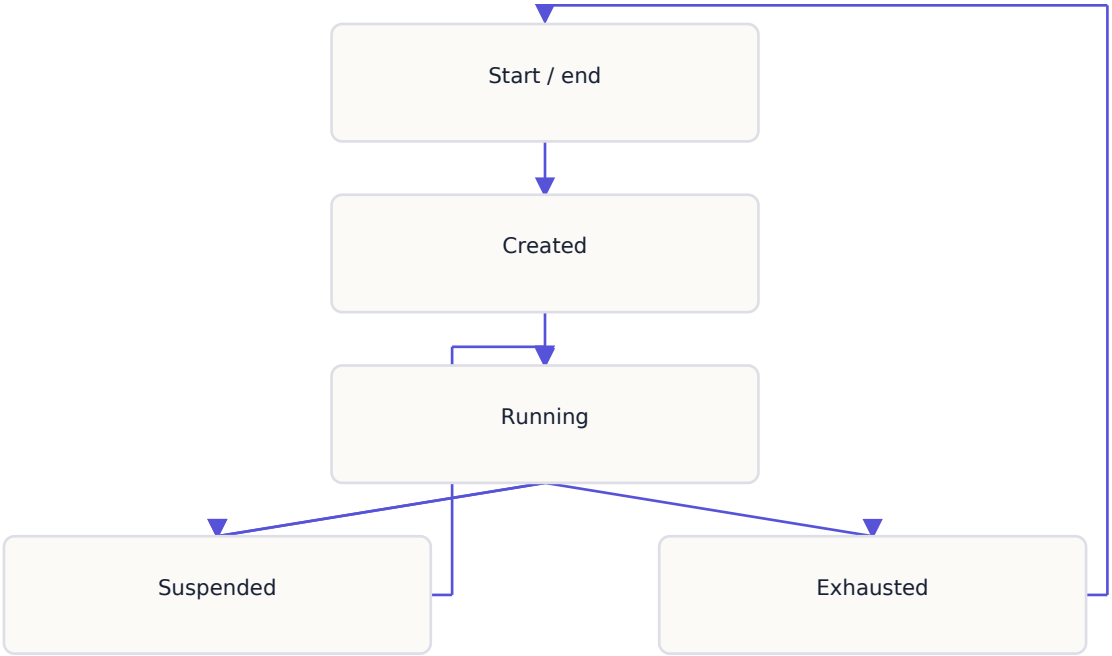
6. `yield` pauses without discarding local state

The next request resumes after the earlier `yield`:

```python
assert next(numbers) == 2
assert next(numbers) == 1
assert next(numbers, "done") == "done"
```

Before yielding `2`, the resumed body executes `start -= 1`, changing the remembered local value from `3` to `2`. A later request repeats that sequence.

One generator object keeps its suspended frame between requests.



Created → Running: first next request

Running → Suspended: yield one value

Suspended → Running: next request resumes

Running → Suspended: yield another value

Running → Exhausted: return or body ends

A normal function loses its ordinary frame after returning. A generator frame is deliberately retained while suspended, including local values and the next instruction position.

7. `return` ends production; `yield` produces a stream item

```python
def limited_steps(limit):
    step = 1
    while step <= limit:
        yield step
        step += 1
    return


assert list(limited_steps(3)) == [1, 2, 3]
assert list(limited_steps(0)) == []
```

Each `yield` contributes an item visible to ordinary iteration. Reaching `return` or the end signals exhaustion. Python generators can attach a value to `StopIteration`, but ordinary `for` and `list(...)` consumption does not treat that value as another yielded item. Foundations code should return only to stop unless a specialized protocol explicitly needs more.

Make finite boundaries clear. This course avoids an unbounded generator unless the consumer has an equally visible limit.

Checkpoint: generator timing

Generator timing checkpoint

Separate object creation, body execution, suspension, and exhaustion.

1. What happens when `numbers = countdown(3)` runs?

- **A.** All values print immediately
- **B.** A generator object is created without starting the body
- **C.** Only the final value is returned
- **D.** The list `[3, 2, 1]` is stored

2. Where is execution suspended after the generator yields 3?

- **A.** Before entering the function
- **B.** After the function frame is deleted
- **C.** At the yield point, before the following decrement
- **D.** Inside the source list

3. Why does the second `next(numbers)` return 2?

- **A.** The first yield changed the source list
- **B.** Python sorts yielded values
- **C.** A new generator starts at 2
- **D.** The suspended frame resumes, decrements its remembered `start`, then yields again

4. What does bare `return` do inside `limited_steps`?

- **A.** Ends production without yielding another item
- **B.** Restarts the generator

- **C.** Returns the generator’s source collection
- **D.** Makes every earlier item reusable

Write your choices before checking the answer key at the end of this book.

8. `yield from` delegates to another iterable

Without delegation, a generator can relay items with a nested loop:

```python
def flatten_groups(groups):
    for group in groups:
        for item in group:
            yield item
```

`yield from` expresses the relay directly:

```python
def flatten_groups(groups):
    for group in groups:
        yield from group


groups = [["red", "blue"], [], ["silver"]]
assert list(flatten_groups(groups)) == ["red", "blue", "silver"]
```

When a group is empty, delegation produces no values and continues. In the recursive `leaf_names` preview, `yield from leaf_names(child)` delegates to a child generator rather than materializing its entire result first.

9. Generator expressions are lazy comprehensions

Square brackets create a list immediately. Parentheses create a generator expression:

```python
readings = [12, 18, 25, 30]

eager = [value * 2 for value in readings]
lazy = (value * 2 for value in readings)

assert eager == [24, 36, 50, 60]
assert next(lazy) == 24
assert list(lazy) == [36, 50, 60]
```

The partial `next` consumed the first lazy result. Converting the same generator to a list continues from its current position; it does not repeat `24`.

Use a list when the complete result will be reused, indexed, or measured. Use an iterator when one-pass incremental consumption is the desired contract.

10. Consumers can stop a lazy source early

`any` and `all` short-circuit. They stop as soon as the answer is known:

```python
def observed(values):
    for value in values:
```

PYTHON (CONTINUED)

```
        print("checked", value)
        yield value


has_large = any(value > 20 for value in observed([4, 25, 100]))
assert has_large is True
```

The generator does not need to check `100` because `25 > 20` already makes `any(...)` true. Similar early stopping happens with `next` and bounded tools such as `itertools.islice`.

Lazy execution changes timing: errors and side effects inside production occur when a consumer requests the relevant value, not necessarily when the generator object is created. Prefer generators that yield data without surprising effects; the `print` above exists only to reveal timing.

11. Build a pipeline from small lazy stages

PYTHON

```
def above_threshold(values, threshold):
    for value in values:
        if value >= threshold:
            yield value


def to_fahrenheit(celsius_values):
    for value in celsius_values:
        yield value * 9 / 5 + 32


source = [12, 18, 25, 30]
usable = above_threshold(source, 18)
converted = to_fahrenheit(usable)

assert next(converted) == 64.4
assert list(converted) == [77.0, 86.0]
```

The first request flows through both stages only until one output is available. The pipeline resumes from its shared positions later. A stage should document that its input is consumed; passing the same iterator elsewhere means those consumers share progress.

12. Shared consumers can surprise each other

PYTHON

```
source = iter([10, 20, 30, 40])

assert next(source) == 10
assert list(source) == [20, 30, 40]
assert next(source, None) is None
```

If two functions receive `source`, whichever consumes first determines what the other sees. To repeat work, keep a restartable collection or call a generator factory again:

PYTHON

```
def reading_source():
    yield from [10, 20, 30, 40]


assert list(reading_source()) == [10, 20, 30, 40]
assert list(reading_source()) == [10, 20, 30, 40]
```

Each function call returns a new generator object with new state.

Checkpoint: pipelines and exhaustion

Lazy pipeline checkpoint

Follow which consumer has requested which values.

1. After `next(lazy)` returns 24, what does `list(lazy)` contain?

- **A.** Every result including 24
- **B.** An error because partial use is forbidden
- **C.** Only the remaining results 36, 50, and 60
- **D.** The original source values

2. Why can `any(value > 20 for value in source)` stop after seeing 25?

- **A.** 25 is the last possible value
- **B.** The source list is deleted
- **C.** Generators always produce two items
- **D.** One true item already determines the result of `any`

3. How can code obtain a fresh pass over values produced by `reading_source`?

- **A.** Call `reading_source()` again to create a new generator
- **B.** Call `next` on the exhausted object
- **C.** Rename the exhausted object
- **D.** Return its previous last item

4. What does `yield from group` do in `flatten_groups`?

- **A.** Yields the group list as one value only
- **B.** Relays each value from that iterable before continuing
- **C.** Sorts the group
- **D.** Restarts all earlier groups

Write your choices before checking the answer key at the end of this book.

13. Lab: stream usable sensor readings

Build a finite source and two lazy stages:

```python
def sensor_readings(values):
    """Yield each supplied sensor value in order."""
    raise NotImplementedError


def valid_readings(values, *, minimum=0, maximum=100):
    """Yield values inside the inclusive supported range."""
    raise NotImplementedError
```

PYTHON (CONTINUED)

```
def changes(values):
    """Yield each difference from the previous value."""
    raise NotImplementedError
```

Use progressive consumption checks:

PYTHON

```
source_values = [-5, 10, 14, 200, 20, 20]
stream = sensor_readings(source_values)

assert iter(stream) is stream
assert next(stream) == -5
assert next(stream) == 10
assert list(stream) == [14, 200, 20, 20]
assert next(stream, "exhausted") == "exhausted"

fresh = sensor_readings(source_values)
usable = valid_readings(fresh, minimum=0, maximum=100)
assert next(usable) == 10
assert list(usable) == [14, 20, 20]
assert list(usable) == []

pipeline = changes(valid_readings(sensor_readings(source_values)))
assert next(pipeline) == 4
assert list(pipeline) == [6, 0]
assert source_values == [-5, 10, 14, 200, 20, 20]
assert list(valid_readings([], minimum=0, maximum=100)) == []
```

`changes` needs one previous valid value before it can yield a difference. An empty or one-item input therefore yields no differences. Do not build hidden intermediate lists in these functions.

Hint: keep only the state each stage needs

The source yields each input directly. The validator yields only values inside both boundaries. In `changes`, obtain the first item with `next(iterator, sentinel)`, then loop over the remaining items, yield `current - previous`, and update `previous`.

Show one complete solution after attempting the lab

PYTHON

```
def sensor_readings(values):
    """Yield each supplied sensor value in order."""
    yield from values


def valid_readings(values, *, minimum=0, maximum=100):
    """Yield values inside the inclusive supported range."""
    for value in values:
        if minimum <= value <= maximum:
            yield value


def changes(values):
    """Yield each difference from the previous value."""
    iterator = iter(values)
    missing = object()
    previous = next(iterator, missing)
    if previous is missing:
        return

    for current in iterator:
        yield current - previous
```

PYTHON (CONTINUED)

```
        previous = current
```

After all assertions pass, predict `list(changes([7]))` and `list(changes([7, 3, 9]))` before running them. Explain why the source list stays unchanged while iterator positions advance.

14. Explain the state owner

For each object in the lab, state:

1. when its generator body first runs;
2. which local values remain during suspension;
3. which consumer requests the next value;
4. whether it can restart; and
5. what proves exhaustion.

Restart the notebook and run the full cell sequence once. Stateful examples are easy to misread when earlier exploratory calls already consumed values.

Key points

Key points

- An iterable can provide iterators; an iterator owns a one-pass consumption position.
- Collections commonly create fresh iterators, while reusing one iterator continues from its current position.
- Calling a generator function creates a generator object without running its body.
- The first request starts the body; `yield` produces a value and preserves the suspended frame for a later request.
- Reaching `return` or the end exhausts a generator permanently.
- Generator expressions and pipelines perform work as consumers request it, so short-circuit consumers can stop early.
- Use a list for reusable materialized results and a generator when one-pass incremental production is the intended contract. :::

References

- Python glossary: iterable and iterator
- Python language reference: the `yield` statement
- Python language reference: generator expressions
- Python built-in function: `next`

Adding Behavior with Decorators

Online lesson and runnable notebook

Build decorators from ordinary function reassignment, then preserve arguments, returned results, metadata, state, configuration, and stacked call order.

- **Level:** Python Foundations · Unit 5
- **Estimated time:** 3.5–5 hours
- **You will learn:** Explain decorator syntax as function transformation and write wrappers that forward calls, preserve results and metadata, retain state, accept configuration, and stack predictably.
- **Practice in:** Google Colab, JupyterLab, or a local editor

1. Repeated behavior can wrap several functions

Suppose several game actions need a small audit message:

```python
def open_gate(code):
    print("calling open_gate")
    result = f"gate {code} opened"
    print("finished open_gate")
    return result


def reveal_map(region):
    print("calling reveal_map")
    result = f"map of {region} revealed"
    print("finished reveal_map")
    return result
```

The useful bodies differ, but the before/after behavior repeats. A decorator is a function transformation: it receives one function and returns replacement behavior, usually a wrapper that calls the original.

This lesson will answer:

- When does the decorator run, and when does the wrapper run?
- How does a wrapper preserve arbitrary call shapes and returned values?
- Why does `functools.wraps` matter?
- In what order do configured or stacked decorators operate?

2. Build a wrapper manually before using `@`

Start with a zero-argument function so the transformation is visible:

```python
def add_log(function):
    def wrapper():
        print(f"calling {function.__name__}")
        result = function()
```

PYTHON (CONTINUED)

```
        print(f"finished {function.__name__}")
        return result

    return wrapper


def find_key():
    return "silver key"


find_key = add_log(find_key)
assert find_key() == "silver key"
```

Read the reassignment in three stages:

1. The original `find_key` function becomes the argument to `add_log`.
2. `add_log` creates and returns `wrapper`, whose closure remembers the original.
3. The name `find_key` is rebound to that wrapper.

Later, calling `find_key()` actually calls `wrapper()`, which calls the original function from its enclosing scope.

3. `@decorator` is transformation syntax

The decorator form expresses the same rebinding:

PYTHON

```
@add_log
def read_compass():
    return "north"


assert read_compass() == "north"
```

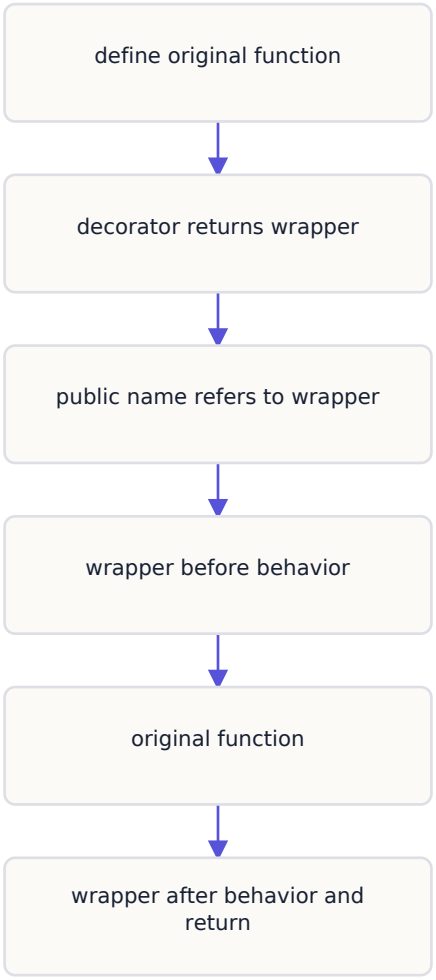
Conceptually, Python performs:

TEXT

```
define the original read_compass function
read_compass = add_log(read_compass)
```

The decorator expression runs when Python executes the `def` statement, often while a module is imported or a notebook definition cell runs. The wrapper body runs only when the decorated function is called later.

Definition time installs replacement behavior; call time travels through the wrapper to the original and back.



public name refers to wrapper → wrapper before behavior: later call

4. A wrapper must return the original result

This broken wrapper calls correctly but discards the result:

PYTHON
```
def broken_log(function):
    def wrapper():
        function()

    return wrapper


@broken_log
def secret_number():
    return 42


assert secret_number() is None
```

The wrapper reaches its end without `return`, so callers see `None` even though the original returned `42`. A transparent decorator must capture and return the original result unless changing that result is its documented purpose.

The required data flow is:

```
PYTHON
result = function()
return result
```

Checkpoint: transformation and timing

Decorator transformation checkpoint

Separate definition-time replacement from later call-time behavior.

1. What ordinary assignment matches `@add_log` above `def read_compass`?

- **A.** `read_compass = add_log(read_compass)`
- **B.** `add_log = read_compass()`
- **C.** `read_compass = None`
- **D.** `wrapper = print`

2. When does `add_log(read_compass)` run?

- **A.** After every return
- **B.** When Python executes the decorated definition
- **C.** Only when the program exits
- **D.** After the wrapper has been exhausted

3. When does the body of the returned `wrapper` run?

- **A.** Before the original is defined
- **B.** During every annotation
- **C.** When callers later call the decorated public name
- **D.** Only when the decorator returns it

4. Why does `secret_number()` return `None` after `broken_log` decorates it?

- **A.** Integers cannot cross wrappers
- **B.** Decorators erase every result
- **C.** The original no longer runs
- **D.** The wrapper calls the original but never returns its result

Write your choices before checking the answer key at the end of this book.

5. Forward every supported argument

The first `add_log` wrapper accepts no arguments. It cannot decorate `open_gate(code)`. Use variadic forwarding:

```
PYTHON
def add_log(function):
    def wrapper(*args, **kwargs):
        print(f"calling {function.__name__}")
        result = function(*args, **kwargs)
        print(f"finished {function.__name__}")
```

PYTHON (CONTINUED)
```
        return result

    return wrapper


@add_log
def open_gate(code, *, quiet=False):
    suffix = " quietly" if quiet else ""
    return f"gate {code} opened{suffix}"


assert open_gate("A7") == "gate A7 opened"
assert open_gate("B2", quiet=True) == "gate B2 opened quietly"
```

`*args` collects positional arguments passed to the wrapper. `**kwargs` collects keyword arguments. The original call `function(*args, **kwargs)` expands both collections, preserving the caller's call shape.

A wrapper that forwards only positional arguments would lose `quiet=True`. A wrapper that forwards arguments but not the result still breaks the return contract. Both directions matter: arguments flow inward; results flow outward.

6. Preserve useful metadata with `functools.wraps`

Without help, the public function reports the wrapper's name and docstring:

PYTHON
```
assert open_gate.__name__ == "wrapper"
```

Tools, help systems, and readers need the original public identity. Apply `functools.wraps` to the wrapper:

PYTHON
```
from functools import wraps


def add_log(function):
    @wraps(function)
    def wrapper(*args, **kwargs):
        print(f"calling {function.__name__}")
        result = function(*args, **kwargs)
        print(f"finished {function.__name__}")
        return result

    return wrapper


@add_log
def reveal_map(region):
    """Return a message revealing one region."""
    return f"map of {region} revealed"


assert reveal_map("Crater") == "map of Crater revealed"
assert reveal_map.__name__ == "reveal_map"
assert reveal_map.__doc__ == "Return a message revealing one region."
```

`wraps(function)` is itself a decorator applied to `wrapper`. It copies key metadata and exposes a `__wrapped__` link used by introspection tools. It does not fix lost arguments or results; your wrapper still owns those responsibilities.

7. A decorator closure can keep focused state

```python
from functools import wraps


def count_calls(function):
    count = 0

    @wraps(function)
    def wrapper(*args, **kwargs):
        nonlocal count
        count += 1
        result = function(*args, **kwargs)
        return count, result

    return wrapper


@count_calls
def collect_star(color):
    return f"{color} star"


assert collect_star("blue") == (1, "blue star")
assert collect_star("gold") == (2, "gold star")
```

The `count` binding belongs to this decorated function's closure. A separately decorated function receives separate state. This decorator intentionally changes the result contract to `(call_number, original_result)`, so its name and documentation should make that choice clear. A transparent audit decorator would instead log elsewhere and return only the original result.

Checkpoint: transparent wrappers

Wrapper contract checkpoint

Follow arguments inward, results outward, and metadata at the public name.

1. Which wrapper call preserves `quiet=True` for the original function?

- **A.** `function()`
- **B.** `function(*args, **kwargs)`
- **C.** `function(args)`
- **D.** `kwargs(function)`

2. What public metadata does `@wraps(function)` help preserve?

- **A.** Every global variable
- **B.** The wrapper's call count
- **C.** The original function's name and docstring
- **D.** All printed output

3. What must a transparent wrapper preserve across the call boundary?

- **A.** Only the original name
- **B.** Only positional arguments

- **C.** Only print order
- **D.** Arguments going in and the original result coming out

4. Where does `count_calls` keep its call count?

- **A.** In an enclosing closure binding updated with `nonlocal`
- **B.** In every caller's global scope
- **C.** Inside the returned string
- **D.** In `functools.wraps`

Write your choices before checking the answer key at the end of this book.

8. A configurable decorator adds one more function layer

Suppose logging should include a chosen label. The outer function receives configuration and returns the actual decorator:

```python
from functools import wraps


def tagged(label):
    def decorate(function):
        @wraps(function)
        def wrapper(*args, **kwargs):
            result = function(*args, **kwargs)
            return "[" + label + "] " + result

        return wrapper

    return decorate


@tagged("TREASURE")
def describe_treasure(name):
    return f"found {name}"


assert describe_treasure("moonstone") == "[TREASURE] found moonstone"
```

Trace the three layers:

Function layer	Receives	Returns
`tagged`	label configuration	`decorate`
`decorate`	original function	`wrapper`
`wrapper`	later call arguments	decorated call result

At definition time, `tagged("TREASURE")` runs first and returns `decorate`. Python then applies that decorator to `describe_treasure`. The wrapper waits for later calls.

9. Stacked decorators apply inside out

```python
from functools import wraps


def surround(left, right):
    def decorate(function):
        @wraps(function)
        def wrapper(*args, **kwargs):
            return left + function(*args, **kwargs) + right

        return wrapper

    return decorate


@surround("<", ">")
@surround("[", "]")
def rune():
    return "moon"


assert rune() == "<[moon]>"
```

Application follows ordinary nested calls:

```text
rune = surround("<", ">")(
    surround("[", "]")(original_rune)
)
```

The decorator closest to `def` wraps the original first. At call time, the outer `< >` wrapper starts first, calls the inner `[ ]` wrapper, which calls the original. Results then return outward. Swapping the decorator lines changes the result to `[<moon>]`.

10. Decorators should make the public contract clearer, not mysterious

Good foundational uses are narrow and visible: logging, counting, timing at a later stage, access checks in frameworks, caching, or registering functions. Every decorator changes the effective public callable, even when it tries to be transparent.

Avoid a decorator when:

- a direct helper call would be clearer;
- the wrapper silently changes the result type;
- decoration depends on surprising changing global state;
- several layers make call order difficult to explain; or
- different functions require incompatible call contracts.

Decorator syntax hides a transformation behind one line. Use it only when that shared transformation is easier to understand than explicit repetition.

11. Lab: track game achievements

Create a configurable audit decorator that records one event per successful call while preserving the decorated function’s behavior and metadata:

PYTHON

```
from functools import wraps


def track_achievement(log, *, badge):
    """Return a decorator that appends successful call records to log."""
    raise NotImplementedError
```

The event must be a dictionary with `badge`, `function`, `args`, `kwargs`, and `result`. Append it after the original returns.

PYTHON

```
events = []


@track_achievement(events, badge="gatekeeper")
def open_gate(code, *, quiet=False):
    """Return an opened-gate message."""
    suffix = " quietly" if quiet else ""
    return f"gate {code} opened{suffix}"


@track_achievement(events, badge="cartographer")
def reveal_map(region):
    """Return a revealed-map message."""
    return f"map of {region} revealed"


assert open_gate("A7") == "gate A7 opened"
assert open_gate("B2", quiet=True) == "gate B2 opened quietly"
assert reveal_map("Crater") == "map of Crater revealed"
assert open_gate.__name__ == "open_gate"
assert open_gate.__doc__ == "Return an opened-gate message."
assert events == [
    {
        "badge": "gatekeeper",
        "function": "open_gate",
        "args": ("A7",),
        "kwargs": {},
        "result": "gate A7 opened",
    },
    {
        "badge": "gatekeeper",
        "function": "open_gate",
        "args": ("B2",),
        "kwargs": {"quiet": True},
        "result": "gate B2 opened quietly",
    },
    {
        "badge": "cartographer",
        "function": "reveal_map",
        "args": ("Crater",),
        "kwargs": {},
        "result": "map of Crater revealed",
    },
]
```

The decorator intentionally mutates the supplied `log`; that effect is explicit in its contract. The wrapped game functions still return their original strings.

Hint: write and test one layer at a time

The outer call remembers `log` and `badge`, then returns `decorate`. `decorate` receives `function` and returns a `@wraps(function)` wrapper. The wrapper forwards arguments, keeps the result, appends one dictionary, and returns the result.

Show one complete solution after attempting the lab

PYTHON

```
from functools import wraps


def track_achievement(log, *, badge):
    """Return a decorator that appends successful call records to log."""
    def decorate(function):
        @wraps(function)
        def wrapper(*args, **kwargs):
            result = function(*args, **kwargs)
            event = {
                "badge": badge,
                "function": function.__name__,
                "args": args,
                "kwargs": kwargs.copy(),
                "result": result,
            }
            log.append(event)
            return result

        return wrapper

    return decorate
```

Checkpoint: configuration and stacking

Configured decorator checkpoint

Count the function layers and trace nested order.

1. What does `track_achievement(events, badge='gatekeeper')` return?

- **A.** The final gate message
- **B.** The actual decorator that will receive a function
- **C.** An already completed event
- **D.** The original `events` list

2. Why does the solution append an event after calling the original?

- **A.** Arguments exist only after return
- **B.** `wraps` requires a list
- **C.** The event can include the actual successful returned result
- **D.** The wrapper cannot run before the original

3. Which decorator wraps the original `rune` first?

- **A.** Both simultaneously
- **B.** The top < > decorator
- **C.** Whichever wrapper prints first

- **D.** The `[ ]` decorator closest to `def`

4. Why is the final rune text `<[moon]>`?

- **A.** The outer `< >` wrapper receives and surrounds the inner `[moon]` result
- **B.** Python sorts punctuation
- **C.** The original function returns brackets
- **D.** Both decorators mutate one global string

Write your choices before checking the answer key at the end of this book.

12. Explain definition time and call time separately

Using `track_achievement`, state:

1. when configuration arguments are evaluated;
2. when the original function is passed to `decorate`;
3. what public name refers to after definition;
4. what the wrapper receives on each later call;
5. where the result travels; and
6. which state changes deliberately.

Then apply `@surround` above a tracked function and predict both the returned text and recorded result. The outer wrapper sees the result of the complete inner decorated call; reversing the stack changes which result the audit records.

Key points

Key points

- Decorator syntax passes a newly defined function to another function and rebinds the public name to the returned callable.
- Decorator application occurs at definition time; wrapper bodies run at later call time.
- Transparent wrappers forward `*args` and `**kwargs` and return the original result.
- `functools.wraps` preserves useful public metadata but does not repair broken forwarding or missing returns.
- Closures can retain small per-decorated-function state; `nonlocal` updates it deliberately.
- A configurable decorator has three layers: configuration, decoration, and later wrapped calls.
- Stacked decorators apply inside out and execute as nested wrappers, with results returning outward. :::

References

- Python glossary: decorator
- Python language reference: function definitions and decorators

- Python standard library: `functools.wraps`

Unit Challenge: Unlock the Moonlit Library

Online lesson and runnable notebook

Traverse a nested magical archive with focused functions, a recursive generator, callback-selected clues, and an audited public unlock operation.

- **Level:** Python Foundations · Unit 5 challenge
- **Estimated time:** 60–120 minutes
- **Outcome assessed:** Compose clear interfaces, recursive lazy traversal, callback policy, and a metadata-preserving decorator into one reusable archive operation.
- **Evidence:** Pass 24 progressive assertions, preserve a debugging record, and explain the derived library message.

1. Open the archive without hard-coding its answer

At moonrise, the Moonlit Library rearranges its shelves. Nine readable scrolls contain numbered letters. Sealed scrolls are decoys, decorative scrolls have no clue, and shelves may contain smaller shelves. Your archive engine must discover the message from the data and return the final artifact:

```text
MOONLIGHT OPENS THE ARCHIVE
```

You will build a small set of cohesive functions rather than one long solution:

1. normalize scroll titles through a keyword-only option;
2. traverse shelves with a recursive generator;
3. select scrolls through a predicate callback;
4. decode ordered clues into one returned word;
5. compose those steps in one public unlock operation; and
6. audit that public call with a decorator that preserves its contract.

Keep the solution closed for the first attempt

Plan for one or two focused hours. Run one assertion group at a time. Open a hint only after recording the exact function, input, actual result, and expected result that currently disagree. :::

2. Read the archive shape and constraints

A node has one of two forms:

- a **shelf** has `kind`, `name`, and `entries`, where every entry is another shelf or scroll;
- a **scroll** has `kind`, `title`, `sealed`, and `clue`, where a clue is either `(position, letter)` or `None`.

The traversal order is depth first and left to right. When the current node is a scroll, yield it. When it is a shelf, recurse into each entry in order.

Constraints

1. Preserve every supplied public name, signature, and docstring.
2. Derive the artifact from the archive; do not assign `"MOONLIGHT"` or the solved sentence directly in a function.
3. Do not mutate the archive or its scroll records.
4. Keep traversal lazy: `iter_scrolls` and `select_scrolls` must yield values, not build hidden result lists.
5. Pass a predicate function into `select_scrolls`; do not hard-code the readable-scroll policy inside traversal.
6. Keep calculation functions free of printing.
7. The decorator must forward arguments, return the original result, preserve metadata with `wraps`, and append one audit event after a successful call.
8. Use only the Python standard library; classes, files, and exception handlers are outside this challenge.

3. Start from the contract

Run the data cell unchanged:

PYTHON

```
archive = {
    "kind": "shelf",
    "name": "Atrium",
    "entries": [
        {
            "kind": "scroll",
            "title": " moon map ",
            "sealed": False,
            "clue": (0, "M"),
        },
        {
            "kind": "shelf",
            "name": "East Gallery",
            "entries": [
                {
                    "kind": "scroll",
                    "title": "Owl Song",
                    "sealed": False,
                    "clue": (1, "O"),
                },
                {
                    "kind": "shelf",
                    "name": "Upper Nook",
                    "entries": [
                        {
                            "kind": "scroll",
                            "title": "Oracle Note",
                            "sealed": False,
                            "clue": (2, "O"),
                        },
                        {
                            "kind": "scroll",
                            "title": "Counterfeit Omen",
                            "sealed": True,
                            "clue": (2, "X"),
                        },
                        {
                            "kind": "scroll",
```

PYTHON (CONTINUED)

```
                            "title": "North Bell",
                            "sealed": False,
                            "clue": (3, "N"),
                        },
                    ],
                },
            ],
        },
        {
            "kind": "shelf",
            "name": "Lantern Garden",
            "entries": [
                {
                    "kind": "scroll",
                    "title": "Lantern Leaf",
                    "sealed": False,
                    "clue": (4, "L"),
                },
                {
                    "kind": "scroll",
                    "title": "Idle Decoration",
                    "sealed": False,
                    "clue": None,
                },
                {
                    "kind": "shelf",
                    "name": "Glass Cabinet",
                    "entries": [
                        {
                            "kind": "scroll",
                            "title": "Ink Index",
                            "sealed": False,
                            "clue": (5, "I"),
                        },
                        {
                            "kind": "scroll",
                            "title": "Glass Glyph",
                            "sealed": False,
                            "clue": (6, "G"),
                        },
                        {
                            "kind": "scroll",
                            "title": "Horizon Hymn",
                            "sealed": False,
                            "clue": (7, "H"),
                        },
                    ],
                },
            ],
        },
        {
            "kind": "scroll",
            "title": "Twilight Key",
            "sealed": False,
            "clue": (8, "T"),
        },
        {
            "kind": "scroll",
            "title": "Sealed Mirror",
            "sealed": True,
            "clue": None,
        },
    ],
}
```

Then copy the scaffold. Implement the functions in this order; do not change the public call shapes.

```python
from functools import wraps


def normalize_title(title, *, separator="-"):
    """Return a case-folded title with words joined by separator."""
    raise NotImplementedError


def iter_scrolls(node):
    """Yield scroll nodes depth first and left to right."""
    raise NotImplementedError


def is_readable(scroll):
    """Return whether a scroll is unsealed and contains a clue."""
    raise NotImplementedError


def select_scrolls(scrolls, predicate):
    """Yield scrolls for which predicate(scroll) is true."""
    raise NotImplementedError


def decode_clues(scrolls):
    """Return clue letters ordered by numeric clue position."""
    raise NotImplementedError


def audit_calls(log):
    """Return a decorator that appends one event after each successful call."""
    raise NotImplementedError


audit_log = []


@audit_calls(audit_log)
def unlock_archive(archive, *, predicate=is_readable):
    """Return the decoded word followed by the archive-opening phrase."""
    raise NotImplementedError
```

An audit event has exactly these fields:

```python
{
    "function": "unlock_archive",
    "args_count": 1,
    "keyword_names": [],
    "result": "MOONLIGHT OPENS THE ARCHIVE",
}
```

`keyword_names` is the sorted list of keyword names received by the wrapper.

4. Build the engine in observable stages

Use this order so one missing contract does not hide another:

1. Normalize ordinary, irregular-space, and empty titles. An empty title becomes `"untitled"`; do not use the separator as a default title.
2. Implement the generator’s scroll base case. Then delegate shelf entries with a recursive call.

3 Create a fresh generator and prove partial consumption before converting any complete traversal to a list.

4 Implement the readable predicate and selector. Keep `predicate(scroll)` in the selector, not inside the traversal function.

5 Extract clue tuples, sort them by numeric position, and join uppercase letters. An empty input returns the empty string.

6 Compose a **fresh** traversal and selection in `unlock_archive`. Reusing the exhausted probe will lose scrolls.

7 Implement the three decorator layers: `audit_calls(log)`, `decorate(function)`, and `wrapper(*args, **kwargs)`.

8 Call the decorated public operation only after all smaller assertions pass.

Write a short trace for the recursive route from `Atrium` to `Oracle Note`. Mark the shelf frame that waits, the smaller entry passed onward, and the scroll that is yielded.

5. Run progressive assertions

The 24 numbered assertions are the acceptance contract. Run each group after implementing its named behavior; do not weaken an expected value.

Title and interface evidence

PYTHON

```
assert callable(normalize_title)                                        # 1
assert normalize_title.__doc__.startswith("Return a case-folded")     # 2
assert normalize_title("  Moon   Map  ") == "moon-map"                # 3
assert normalize_title("Moon Map", separator="_") == "moon_map"      # 4
assert normalize_title("   ") == "untitled"                           # 5
```

Recursive traversal and source evidence

PYTHON

```
import copy

before = copy.deepcopy(archive)
all_scrolls = list(iter_scrolls(archive))

assert len(all_scrolls) == 12                                           # 6
assert all_scrolls[0]["title"] == " moon map "                          # 7
assert all_scrolls[-1]["title"] == "Sealed Mirror"                    # 8
assert archive == before                                                # 9
```

Partial consumption and clean exhaustion

PYTHON

```
probe = iter_scrolls(archive)
assert next(probe)["title"] == " moon map "                             # 10
assert next(probe)["title"] == "Owl Song"                             # 11
assert len(list(probe)) == 10                                           # 12
assert next(probe, "exhausted") == "exhausted"                        # 13
```

Callback and decoding evidence

```python
readable = list(select_scrolls(iter_scrolls(archive), is_readable))

assert len(readable) == 9                                              # 14
assert all(not scroll["sealed"] and scroll["clue"] for scroll in readable)  # 15
assert decode_clues(readable) == "MOONLIGHT"                           # 16
assert decode_clues([]) == ""                                         # 17
```

Decorated composition evidence

```python
assert unlock_archive.__name__ == "unlock_archive"                     # 18
assert unlock_archive.__doc__.startswith("Return the decoded word")   # 19

artifact = unlock_archive(archive)
assert artifact == "MOONLIGHT OPENS THE ARCHIVE"                       # 20
assert len(audit_log) == 1                                             # 21
assert audit_log[0] == {                                               # 22
    "function": "unlock_archive",
    "args_count": 1,
    "keyword_names": [],
    "result": "MOONLIGHT OPENS THE ARCHIVE",
}
```

Controlled variation evidence

Seal the first real clue and add an empty shelf. The decoder now returns the remaining ordered letters, and the recursive generator must cross the empty shelf without yielding a value.

```python
changed_archive = copy.deepcopy(archive)
changed_archive["entries"][0]["sealed"] = True
changed_archive["entries"].append(
    {"kind": "shelf", "name": "Silent Annex", "entries": []}
)

changed_artifact = unlock_archive(changed_archive)
assert changed_artifact == "OONLIGHT OPENS THE ARCHIVE"                # 23
assert len(audit_log) == 2 and audit_log[-1]["result"] == changed_artifact  # 24
```

Do not turn an exhausted generator into a decoding bug

If a complete result is missing its first letters, check whether exploratory `next(...)` calls consumed the same generator. Create a fresh generator for each independent complete traversal. :::

6. Use the hint ladder only when needed

Hint 1: locate the contract that currently disagrees

Write down the function name, input shape, expected returned value, and intended side effect. A scroll is the generator’s base case. A shelf owns recursive progress through its smaller `entries`. Title normalization can split on whitespace and join the words without changing the source string.

Hint 2: follow the boundary between behaviors

`iter_scrolls` knows nothing about sealed clues. `select_scrolls` calls the supplied predicate for every yielded scroll. `decode_clues` receives only the selected scrolls, extracts their clue tuples, sorts by position, and joins the letters. Use a fresh traversal when composing those stages.

Hint 3: assemble the decorator and public pipeline

Pseudocode for the public operation:

```text
scroll stream = recursively traverse archive
readable stream = select from scroll stream with predicate
word = decode readable stream
return word + opening phrase
```

The decorator factory returns `decorate`; `decorate` returns a `@wraps(function)` wrapper; the wrapper calls with `*args, **kwargs`, stores the result, appends the specified event, and returns the unchanged result.

7. Keep debugging evidence

Preserve one failed assertion from before your repair. Do not record only “it did not work.” Connect the evidence to one call boundary or state transition.

Failure	Arguments and actual result	Expected contract	Single hypothesis	Controlled change and rerun
Which numbered assertion failed?	What exact values or error appeared?	What should this function yield, return, preserve, or append?	Which base case, callback, generator state, or wrapper step could explain it?	What one edit did you make, and which earlier checks still pass?

A useful record might reveal that a probe had already consumed two scrolls, that the predicate was called too early, or that the wrapper forgot to return the original result. Keep the record even after the complete suite passes.

8. Compare with a complete solution

Open this only after making a serious attempt and using the hints in order.

Show one complete Moonlit Library solution

```python
from functools import wraps


def normalize_title(title, *, separator="-"):
    """Return a case-folded title with words joined by separator."""
    words = title.casefold().split()
    if not words:
        return "untitled"
    return separator.join(words)


def iter_scrolls(node):
    """Yield scroll nodes depth first and left to right."""
    if node["kind"] == "scroll":
        yield node
        return

    for entry in node["entries"]:
        yield from iter_scrolls(entry)


def is_readable(scroll):
    """Return whether a scroll is unsealed and contains a clue."""
```

PYTHON (CONTINUED)

```
    return not scroll["sealed"] and scroll["clue"] is not None


def select_scrolls(scrolls, predicate):
    """Yield scrolls for which predicate(scroll) is true."""
    for scroll in scrolls:
        if predicate(scroll):
            yield scroll


def decode_clues(scrolls):
    """Return clue letters ordered by numeric clue position."""
    clues = []
    for scroll in scrolls:
        clues.append(scroll["clue"])
    clues.sort(key=lambda clue: clue[0])
    return "".join(letter.upper() for _, letter in clues)


def audit_calls(log):
    """Return a decorator that appends one event after each successful call."""
    def decorate(function):
        @wraps(function)
        def wrapper(*args, **kwargs):
            result = function(*args, **kwargs)
            event = {
                "function": function.__name__,
                "args_count": len(args),
                "keyword_names": sorted(kwargs),
                "result": result,
            }
            log.append(event)
            return result

        return wrapper

    return decorate


audit_log = []


@audit_calls(audit_log)
def unlock_archive(archive, *, predicate=is_readable):
    """Return the decoded word followed by the archive-opening phrase."""
    scrolls = iter_scrolls(archive)
    selected = select_scrolls(scrolls, predicate)
    word = decode_clues(selected)
    return f"{word} OPENS THE ARCHIVE"
```

9. Predict an alternate library policy

The supplied `is_readable` rejects sealed clues. Try an alternate predicate that accepts every scroll containing a clue:

PYTHON

```
def has_clue(scroll):
    return scroll["clue"] is not None
```

Before calling `unlock_archive(archive, predicate=has_clue)`, predict:

- whether the sealed `Counterfeit Omen` enters the selected stream;
- where its `(2, "X")` clue appears after stable sorting;

- the exact decoded word;
- the new `keyword_names` audit value; and
- whether the original archive changes.

The extra clue should make the danger of changing policy visible, not merely rename the same result. Explain why traversal required no edits.

10. Check your understanding

Answer from the completed artifact and its traces rather than from the story alone.

Moonlit Library evidence check

Follow the recursive stream, callback policy, and decorated result.

1. Why must `unlock_archive` create a fresh `iter_scrolls(archive)` stream?

- **A.** Fresh generators sort dictionaries
- **B.** A generator can contain only one scroll
- **C.** Decorators require new global data
- **D.** An earlier partially or fully consumed generator would resume late or be exhausted

2. What is the base case of `iter_scrolls`?

- **A.** A scroll node is yielded without descending into entries
- **B.** A shelf sorts all clues
- **C.** The audit log has one event
- **D.** The title contains a space

3. Which component decides whether a sealed clue is accepted?

- **A.** Title normalization
- **B.** The predicate passed through `select_scrolls`
- **C.** The recursive shelf traversal
- **D.** `functools.wraps`

4. What must the audit wrapper do after appending its event?

- **A.** Return the audit list instead
- **B.** Call the original a second time
- **C.** Return the original artifact unchanged
- **D.** Exhaust every earlier generator again

Write your choices before checking the answer key at the end of this book.

11. Decide whether the challenge is complete

Evidence rubric

Evidence	Ready to record when
Function contracts	Every supplied name, signature, docstring, result, and deliberate side effect matches its promise.
Recursive and lazy behavior	You can trace a nested shelf, partial consumption, resume, and exhaustion.
Callback policy	You can explain why changing the predicate changes selection without changing traversal.
Decorated behavior	Arguments, result, metadata, and one audit event survive the wrapper.
Debugging	One record connects a failed assertion to a specific state or boundary and verified rerun.
Reproducibility	All 24 assertions pass from a clean notebook state with the solution section closed.

Record completion only when every statement is true:

- ☐ All 24 supplied assertions pass without editing expected values.
- ☐ The final artifact is derived from archive data rather than assigned directly.
- ☐ The original archive remains unchanged.
- ☐ I can explain downward recursive calls and upward yielded/returned values.
- ☐ I tested one alternate predicate and predicted its effect before running it.
- ☐ I kept one complete debugging record.
- ☐ The entire artifact works after restarting and running all cells in order.

This button stores a self-reported marker only in this browser. It does not submit the artifact, grade it, verify identity, or issue a certificate.

Key points

- Focused contracts make a large challenge solvable one returned or yielded value at a time.
- Recursive traversal follows nested data; generators expose that traversal incrementally.
- Callback policy belongs outside the traversal, so the same mechanism supports controlled selection changes.
- Fresh generator objects prevent exploratory consumption from contaminating a complete result.
- A transparent audit decorator forwards arguments, preserves metadata, records one event, and returns the original artifact.
- The successful sentence is evidence produced by composition, not a value hidden directly in the implementation.

Unit 6: Sharing, Changing, and Copying Objects

Online lesson and runnable notebook

Learn to see which names and containers reach the same objects, then choose intentional sharing, mutation, or copying from a clear ownership contract.

- **Level:** Python Foundations · Unit 6
- **Estimated time:** 45–75 minutes for orientation and readiness work
- **You will learn:** Trace shared objects, ask who owns future changes, and follow the four-lesson path from identity to copy depth and mutation-aware function design.
- **Practice in:** Google Colab, JupyterLab, or a local editor

1. One update appears in two places

Two explorers appear to have separate inventory names:

```python
nova_inventory = ["lantern", "rope"]
sol_inventory = nova_inventory

sol_inventory.append("moon key")

assert nova_inventory == ["lantern", "rope", "moon key"]
assert sol_inventory == ["lantern", "rope", "moon key"]
```

There is only one list. Assignment did not repeat the list construction; it bound `sol_inventory` to the object already reached through `nova_inventory`. Appending through either name changes that shared list.

Now compare equal independent values:

```python
nova_inventory = ["lantern", "rope"]
sol_inventory = ["lantern", "rope"]

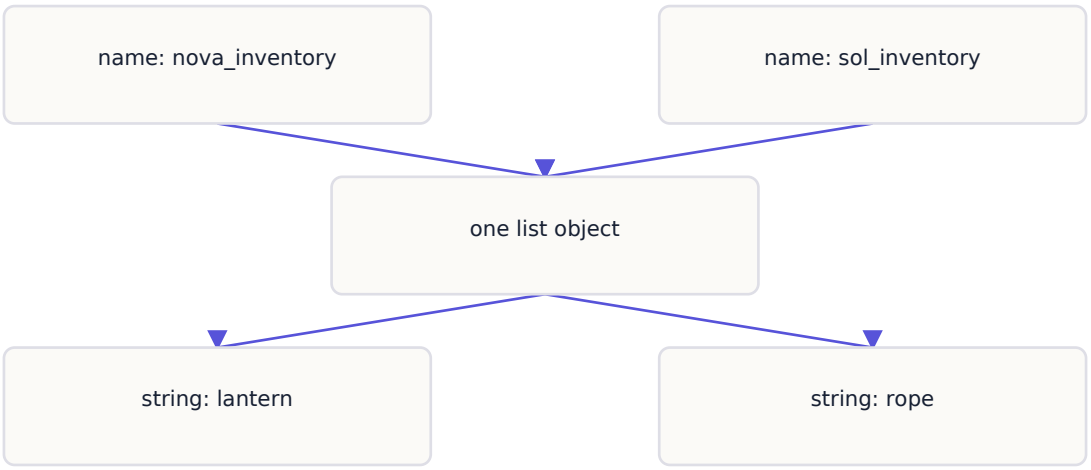
assert nova_inventory == sol_inventory
assert nova_inventory is not sol_inventory
```

The lists contain equal values, but they are different objects. A mutation of one does not change the other. Unit 6 gives you the language and evidence to tell these situations apart before a surprising edit reaches the wrong data.

2. Names point; objects carry identity and state

Use a small diagram instead of imagining variables as sealed boxes. Names are ways to reach objects. Containers also hold references to other objects.

Both inventory names point to one mutable list. The list contains references to two string objects.



Every object has a type, a value or observable state, and an identity. The identity answers “is this the very same object?” It is distinct from asking whether two objects currently compare equal.

When code runs, watch two kinds of change:

- **rebinding** changes which object a name reaches;
- **mutation** changes the state of an existing object.

Those events can produce similar printed output while giving later code very different behavior.

3. Ask four questions before editing data

Before modifying a collection or passing it to a helper, ask:

1. **Are the values equal?** Use `==` for ordinary value comparison.
2. **Are they the same object?** Use `is` only when identity is the question.
3. **Can this object change in place?** Lists, dictionaries, and sets commonly can; strings, numbers, and tuples cannot change their own stored contents.
4. **Who should observe future changes?** This ownership decision determines whether sharing, mutation, or copying is appropriate.

The last question prevents automatic rules such as “never mutate” or “always deep copy.” Shared state can be intentional. A scoreboard may need one list that several views observe. A new player’s inventory may need independent nested state. Correctness depends on the contract.

4. Choose a sharing policy, not a reflex

Four broad choices will recur:

Choice	What stays shared?	Suitable when
Keep the reference	the complete object graph	every user should observe the same updates
Shallow copy	nested objects, but not the outer container	only outer membership or fields will change

Choice	What stays shared?	Suitable when
Selective copy	only fields named by the ownership design	known mutable descendants need independence
Deep copy	a recursively copied supported graph	the complete mutable graph must be independently editable

The most expensive or deepest choice is not automatically the safest design. Copying can hide intended communication, duplicate large structures, and obscure who owns the authoritative state. First draw which objects may change; then separate only the boundaries that require independent ownership.

5. Your path through this unit

Lesson	Capability you will build	Main lab artifact
1	When Names Share the Same Object — follow bindings and container references, distinguish equality from identity, and use the `None` sentinel correctly.	Party-roster object graph
2	Changing Objects and Choosing Safe Keys — distinguish mutation from rebinding, read method return contracts, and choose hashable dictionary/set keys.	Expedition-state tracker
3	Copying Nested Data Without Surprises — diagnose repeated references and select shallow, selective, or deep copying from ownership needs.	Independent mission boards
4	Designing Functions That Control Mutation — make caller-visible effects, returned copies, defaults, snapshots, and retained state explicit.	Quest-state API

The Unit Challenge asks you to break a Mirror Vault whose doppelgängers accidentally share inventories, journals, and visited-room state.

6. Unit 5 supplied the call boundaries

You already know how to design parameters, returned results, side effects, defaults, closures, and decorated calls. Unit 6 explains which objects cross those boundaries.

PYTHON

```
def add_badge_in_place(badges, badge):
    """Append badge to caller-owned badges; return None."""
    badges.append(badge)


earned = ["starter"]
result = add_badge_in_place(earned, "pathfinder")
```

PYTHON (CONTINUED)

```
assert earned == ["starter", "pathfinder"]
assert result is None
```

The parameter `badges` is a local name, but it reaches the list supplied by the caller. Mutating that object is visible outside the function. Rebinding the local parameter would be different. Lesson 4 develops both traces and turns them into clear API choices.

7. Unit 7 will turn requirements into algorithms

This unit begins with an already-understood task and object graph. It asks how state is shared and who should control changes. Unit 7 adds ambiguous requirements, decomposition, algorithm selection, correctness arguments, and practical efficiency.

You will discuss the cost of copying qualitatively here, because it affects the ownership decision. Formal complexity and evidence-driven algorithm comparison remain in the next unit.

8. Set up an object laboratory

Separate fixture construction, relationships, mutations, and evidence:

PYTHON

```
# 1. Construct source objects.
source = {"name": "Nova", "badges": ["starter"]}

# 2. Establish a relationship deliberately.
alias = source

# 3. Record the expected relationship before mutation.
assert alias is source
assert alias["badges"] is source["badges"]

# 4. Perform one mutation.
alias["badges"].append("pathfinder")

# 5. Check values and relationships.
assert source["badges"] == ["starter", "pathfinder"]
```

Keep value assertions and identity assertions separate. `==` can show that data matches while missing whether a later edit will leak. `is` can show shared identity while saying nothing about whether the current values are correct.

Restart the notebook before interpreting identity-sensitive examples. Old mutations and reused names can make a correct diagram appear wrong.

9. Choose a realistic pace

Unit 6 is planned for approximately **15–23 hours**, including orientation, examples, diagrams, modifications, checkpoints, four final labs, and the challenge. The range is guidance rather than a deadline.

A useful rhythm is:

1. draw names and container paths before running;
2. predict both value equality and object identity;

3 perform one rebind, mutation, or copy;

4 update the graph rather than drawing a disconnected new picture;

5 answer a checkpoint from the graph;

6 finish the lab before opening its hint or solution; and

7 restart before checking repeated calls or old shared state.

10. Check your readiness

Predict every assertion before running:

```python
first = ["map"]
alias = first
equal_copy = ["map"]

assert first == alias
assert first is alias
assert first == equal_copy
assert first is not equal_copy

alias.append("key")
assert first == ["map", "key"]
assert equal_copy == ["map"]

alias = ["compass"]
assert first == ["map", "key"]
assert alias == ["compass"]
```

Explain two moments separately. `append` changed the list reached through both aliased names. The later assignment changed only the `alias` binding; it did not undo or replace the object still reached through `first`.

You are ready for Lesson 1 when you can draw those relationships and explain why equal lists need not be identical.

Key points

Key points

- Names and container entries provide paths to objects; assignment can create another path without copying the object.
- Equality asks whether values compare equal; identity asks whether two expressions reach the same object.
- Rebinding changes a name-to-object relationship; mutation changes an existing object’s state.
- Copy depth should follow an ownership contract, not an automatic preference for the deepest copy.
- Unit 5 supplied function boundaries; this unit explains what can remain shared across them. :::

When Names Share the Same Object

Online lesson and runnable notebook

Follow names and container entries to objects, distinguish equality from identity, and use identity evidence without relying on implementation-specific object reuse.

- **Level:** Python Foundations · Unit 6
- **Estimated time:** 3–4.5 hours
- **You will learn:** Trace bindings and references, distinguish rebinding from mutation and equality from identity, use `is None`, and repair accidental sharing in a small object graph.
- **Practice in:** Google Colab, JupyterLab, or a local editor

1. Two inventory names can lead to one list

```python
nova_gear = ["lamp", "rope"]
sol_gear = nova_gear

sol_gear.append("key")

assert nova_gear == ["lamp", "rope", "key"]
assert sol_gear == ["lamp", "rope", "key"]
```

There are two names but one list. The assignment `sol_gear = nova_gear` evaluates the expression on the right and binds another name to that same object. It does not run the list literal again or copy its contents.

This lesson answers four questions that printed values alone cannot settle:

- Which object does each name or container position reach?
- Did an operation rebind a name or mutate an existing object?
- Do two values compare equal, or are they the same object?
- When is identity the correct test rather than value equality?

2. A binding connects a name to an object

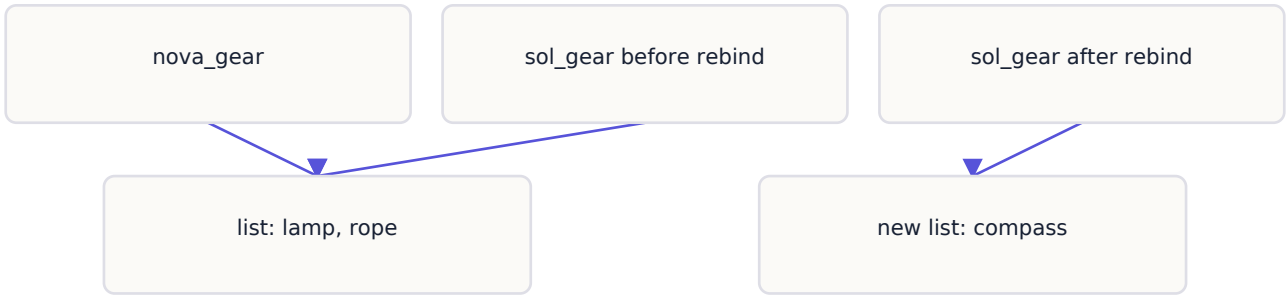
Use these terms consistently:

Term	Meaning in this lesson
object	a Python value with a type, state/value, and identity
name	an identifier such as `nova_gear` in a namespace
binding	the relationship from a name to an object
reference	a way for a name or container entry to reach an object

Term	Meaning in this lesson
alias	another reference to an object that is already reachable

The list literal creates a list object. The first assignment binds `nova_gear`. The second assignment adds the alias `sol_gear`.

Both names reach one list. Rebinding `sol_gear` later changes its arrow, not the old list or `nova_gear`.



The arrows are a learning representation, not syntax Python stores in your source file. They help predict which path will observe a later mutation.

3. Assignment can rebind one name

Continue from one shared list, then rebind only `sol_gear`:

```python
nova_gear = ["lamp", "rope"]
sol_gear = nova_gear

sol_gear = ["compass"]

assert nova_gear == ["lamp", "rope"]
assert sol_gear == ["compass"]
assert nova_gear is not sol_gear
```

The new list literal creates a second list. Assignment changes the binding for `sol_gear`. It does not reach backward through the old binding and replace what `nova_gear` means.

Compare the two events:

Event	Binding changes?	Existing list state changes?
`sol_gear.append("key")`	no	yes
`sol_gear = ["compass"]`	yes	no

This distinction remains important inside functions, loops, and object attributes. Lesson 4 applies it to parameter names.

4. Equality asks about values; identity asks about the object

Create one alias and one equal independent list:

```python
first = ["map", "key"]
alias = first
independent = ["map", "key"]

assert first == alias
assert first is alias

assert first == independent
assert first is not independent
```

`==` delegates to the values' equality behavior. For lists, corresponding elements are compared. `is` does not compare contents; it asks whether both expressions produce the same object.

Four combinations are useful to reason about:

Relationship	`==`	`is`	Example
same object, equal to itself	`True`	`True`	`first` and `alias`
distinct objects, equal contents	`True`	`False`	`first` and `independent`
distinct objects, unequal contents	`False`	`False`	`[1]` and `[2]`
same object but unequal to itself	unusual	`True`	specialized values such as NaN require separate care

The final row is a reminder that equality is behavior defined by a type; it is not the definition of identity. Ordinary beginner data will mostly use the first three rows.

Checkpoint: binding, rebinding, and mutation

Bindings and shared objects checkpoint

Follow each name to an object before predicting the update.

1. What does `sol_gear = nova_gear` do when nova_gear refers to a list?

- **A.** Binds sol_gear to the same list object
- **B.** Recursively copies the list
- **C.** Converts the list to a tuple
- **D.** Removes the nova_gear binding

2. Why does appending through one alias appear through the other?

- **A.** Every list is global
- **B.** Both names reach the one list whose state changed
- **C.** Append creates matching copies
- **D.** Equality forces synchronization

3. What changes in `sol_gear = ['compass']`?

- **A.** The old list becomes a string
- **B.** Every alias is rebound
- **C.** The sol_gear binding now reaches a new list
- **D.** The nova_gear name is deleted

4. Which evidence describes two independently constructed lists with the same contents?

- **A.** They must be unequal and identical
- **B.** They must be unequal aliases
- **C.** They must share every nested object
- **D.** They can satisfy `left == right` and `left is not right`

Write your choices before checking the answer key at the end of this book.

5. Use `id` as temporary diagnostic evidence

`id(object)` returns an integer identifying that object during its lifetime:

```python
items = ["map"]
alias = items
independent = ["map"]

assert id(items) == id(alias)
assert id(items) != id(independent)
```

The exact integers do not matter. Do not treat them as memory addresses promised by the Python language, sort them for meaning, save them as permanent record IDs, or compare them across program runs. After an object no longer exists, an implementation may reuse its old ID for a later object.

For ordinary code, `left is right` communicates an identity question more directly. `id` is helpful in an evidence table or diagram when several paths must be compared at once.

6. `None` is a singleton sentinel

Python has one `None` object. It represents the intentional absence of a value in many APIs:

```python
leader = None

if leader is None:
    message = "Choose a party leader"
else:
    message = f"Leader: {leader}"

assert message == "Choose a party leader"
```

Use `is None` or `is not None`. This is an identity question about a documented singleton, not ordinary content equality.

Do not generalize that style to strings or numbers:

```python
answer = "moon"
assert answer == "moon"

attempts = 3
assert attempts == 3
```

Those are value questions, so `==` is the meaningful operator.

7. Do not learn equality from string or integer reuse

Python implementations may reuse some immutable objects. A literal experiment can therefore make `is` appear to compare values:

```python
left = "".join(["moon", "light"])
right = "moonlight"

assert left == right
```

Do not add an assertion about `left is right`. Its result is not the value contract and can vary with implementation, optimization, or construction path. The same warning applies to small integers and some tuples or constants.

The robust rule is simple:

- use `==` when the program cares whether ordinary values are equal;
- use `is` when the program explicitly cares about the same object, especially `None` or a private sentinel created for that purpose.

Checkpoint: equality, identity, and sentinels

Equality and identity checkpoint

Choose the operator that asks the program’s real question.

1. Which expression asks whether two route lists contain equal values?

- **A.** `left is right`
- **B.** `left == right`
- **C.** `id(left) > id(right)`
- **D.** `left = right`

2. Which check follows Python convention for an absent leader?

- **A.** `leader == 'None'`
- **B.** `id(leader) == 0`
- **C.** `leader is None`
- **D.** `leader = None`

3. Which use of `id()` is reliable for this lesson?

- **A.** Persist it as a database key

- **B.** Interpret it as a portable memory address
- **C.** Compare it across separate runs
- **D.** Compare two live references during one trace

4. Why should code not use `is` to compare ordinary strings?

- **A.** Object reuse is an implementation detail, while the program needs value equality
- **B.** Strings never have identities
- **C.** `is` converts strings to lists
- **D.** Only empty strings support equality

Write your choices before checking the answer key at the end of this book.

8. Containers hold references to their elements

Sharing can happen without two top-level names:

```python
shared_badges = ["starter"]
party = [
    {"name": "Nova", "badges": shared_badges},
    {"name": "Sol", "badges": shared_badges},
]

assert party[0] is not party[1]
assert party[0]["badges"] is party[1]["badges"]

party[0]["badges"].append("vault scout")
assert party[1]["badges"] == ["starter", "vault scout"]
```

The two dictionaries are distinct, but their `"badges"` entries reach one list. An outer identity check cannot answer every nested-sharing question. Follow the complete access path to the object that will mutate.

This explains why nested diagrams matter. A program can have independent outer records and still leak edits through a shared descendant.

9. Rebinding one reference does not remove the others

```python
inventory = ["lamp"]
backup_name = inventory

del inventory

assert backup_name == ["lamp"]
```

`del inventory` removes that name from the current namespace. It does not destroy the list while `backup_name` still reaches it. Similarly, replacing one dictionary entry leaves an old object available through any other references.

Python automatically manages object lifetime. The language does not require you to count references manually, and this unit does not depend on CPython’s reference-counting details. The useful program-level question is whether any reachable path still leads to the object.

10. Repair sharing only when it is accidental

This construction gives every record the same badges list:

```python
template_badges = ["starter"]
players = []

for name in ["Nova", "Sol"]:
    players.append({"name": name, "badges": template_badges})

assert players[0]["badges"] is players[1]["badges"]
```

If badges belong to each player, construct a list for each iteration:

```python
players = []

for name in ["Nova", "Sol"]:
    players.append({"name": name, "badges": ["starter"]})

assert players[0]["badges"] == players[1]["badges"]
assert players[0]["badges"] is not players[1]["badges"]
```

If the shared list represents one team scoreboard that everyone should observe, the first graph may be correct. Identity is not itself a bug. A mismatch between actual sharing and intended ownership is the bug.

11. Lab: map a party roster

Implement helpers that report relationships and build independent player records:

```python
def relationship(left, right):
    """Return equal and identical evidence for two values."""
    raise NotImplementedError


def build_player(name, starting_badges, *, leader=None):
    """Return a player with an independent badges list and optional leader."""
    raise NotImplementedError


def add_badge(player, badge):
    """Append one badge to player and return None."""
    raise NotImplementedError
```

Use these acceptance checks:

```python
starting_badges = ["starter"]
nova = build_player("Nova", starting_badges)
sol = build_player("Sol", starting_badges, leader="Nova")

assert relationship(nova["badges"], sol["badges"]) == {
    "equal": True,
    "identical": False,
}
assert relationship(nova, nova) == {"equal": True, "identical": True}
assert nova["leader"] is None
assert sol["leader"] == "Nova"
```

PYTHON (CONTINUED)

```
result = add_badge(nova, "vault scout")
assert result is None
assert nova["badges"] == ["starter", "vault scout"]
assert sol["badges"] == ["starter"]
assert starting_badges == ["starter"]
assert nova is not sol
```

Before implementing, draw three list objects: the source badges, Nova's badges, and Sol's badges. Equal initial contents do not require shared identity.

Hint: construct the owned list at the player boundary

`relationship` returns the results of `==` and `is`. In `build_player`, create a new outer dictionary and use `list(starting_badges)` for the list that the new player owns. `add_badge` intentionally mutates that one nested list.

Show one complete solution after attempting the lab

PYTHON

```
def relationship(left, right):
    """Return equal and identical evidence for two values."""
    return {"equal": left == right, "identical": left is right}


def build_player(name, starting_badges, *, leader=None):
    """Return a player with an independent badges list and optional leader."""
    return {
        "name": name,
        "badges": list(starting_badges),
        "leader": leader,
    }


def add_badge(player, badge):
    """Append one badge to player and return None."""
    player["badges"].append(badge)
```

Checkpoint: nested reference repair

Party roster checkpoint

Use the lab's value and identity assertions together.

1. What must be compared to learn whether Nova and Sol share their badge list?

- **A.** `nova is sol` only
- **B.** Their name strings
- **C.** `nova['badges'] is sol['badges']`
- **D.** The number of dictionary keys

2. Why does `starting_badges` remain unchanged after adding Nova's badge?

- **A.** Strings protect every list
- **B.** Append creates a tuple
- **C.** Leaders prevent mutation
- **D.** build_player constructed a separate list owned by Nova

3. Which assertion checks the intentional absence of Nova's leader?

- **A.** `nova['leader'] is None`
- **B.** `nova['leader'] == 'None'`
- **C.** `id(nova) == 0`
- **D.** `nova is sol`

4. What does `add_badge` promise in addition to its `None` result?

- **A.** It rebinds every player name
- **B.** It mutates the supplied player's badge list
- **C.** It deep-copies the party
- **D.** It changes the starting list

Write your choices before checking the answer key at the end of this book.

12. Explain the graph after every event

Use your completed lab to answer:

1. Which object does each outer player name reach?
2. Why do the two badges lists compare equal before mutation?
3. Which assertion proves they are independently owned?
4. What changes when `add_badge` runs: a name, an outer dictionary, or a nested list's state?
5. Why is `leader is None` a different kind of question from `leader == "Nova"`?

Then deliberately make both players share one badges list, predict the leak, observe it once, and restore the independent construction. The failure is useful evidence only when you can point to the exact shared path.

Key points

Key points

- Assignment binds a name to an evaluated object; it does not copy that object.
- Rebinding redirects a name. Mutation changes an object seen through every reference that still reaches it.
- `==` asks about value equality; `is` asks about object identity.
- Use `is None` for the singleton sentinel and `==` for ordinary string, number, and collection values.
- Treat `id()` as opaque temporary diagnostic evidence and never rely on literal interning for program correctness.
- Containers hold references, so independent outer objects may still share a nested mutable object.
- Sharing is correct when it matches ownership; repair only accidental sharing. :::

References

- Python language reference: objects, values, and types
- Python language reference: identity comparisons
- Python built-in function: `id`
- PEP 8: comparisons to singletons such as `None`

Changing Objects and Choosing Safe Keys

Online lesson and runnable notebook

Predict which built-in operations mutate existing objects or return new values, then choose dictionary keys and set members from stable hash behavior.

- **Level:** Python Foundations · Unit 6
- **Estimated time:** 3.5–5 hours
- **You will learn:** Distinguish mutation from rebinding, predict common method and `+=` behavior, explain nested mutability, and choose hashable values for dictionary and set lookup.
- **Practice in:** Google Colab, JupyterLab, or a local editor

1. One operation changes an object; another returns a replacement

PYTHON

```
supplies = ["water", "map"]
alias = supplies
mission_name = "moon trail"

append_result = supplies.append("rope")
upper_name = mission_name.upper()

assert append_result is None
assert supplies == ["water", "map", "rope"]
assert alias == ["water", "map", "rope"]
assert mission_name == "moon trail"
assert upper_name == "MOON TRAIL"
```

`list.append` mutates the existing list. Its alias observes the change. Strings are immutable: `str.upper` cannot change the string object, so it returns another string value. The original name still reaches the original lowercase value.

This lesson answers:

- Which common objects can change in place?
- Why do many mutating methods return `None`?
- Why can `+=` mutate one type and rebind another?
- What makes a value suitable for dictionary or set lookup?

2. Mutation preserves identity while changing state

```python
route = ["gate", "bridge"]
before_id = id(route)

route.append("tower")

assert id(route) == before_id
assert route == ["gate", "bridge", "tower"]
```

The list's observable contents changed while its identity remained the same. Every reference to that list can observe the updated state.

Rebinding produces a different event:

```python
route = ["gate", "bridge"]
before = route

route = route + ["tower"]

assert route == ["gate", "bridge", "tower"]
assert before == ["gate", "bridge"]
assert route is not before
```

List concatenation built a new list, then assignment rebound `route`. The name `before` still reaches the original object.

3. Common built-in types have different mutation contracts

Use this table as a guide, then read the documentation for the exact operation:

Common type	Can that object change in place?	Typical transformation
`list`	yes	append, item assignment, `sort`
`dict`	yes	item assignment, `update`, `pop`
`set`	yes	add, `discard`, `update`
`bytearray`	yes	item assignment, `extend`
`int`, `float`, `complex`, `bool`	no	arithmetic returns a value
`str`, `bytes`	no	methods return a value
`tuple`, `frozenset`	no	operations return a value
`None`	no	used as a singleton sentinel

User-defined instances are generally mutable unless their design prevents updates. Unit 11 develops classes and custom equality/hash behavior. For now, make decisions with built-in values whose contracts are documented.

"Immutable" does not mean a name can never change. A name may be rebound to a new immutable object:

```python
attempts = 2
old_attempts = attempts
attempts = attempts + 1

assert attempts == 3
assert old_attempts == 2
```

The integer `2` did not become `3`; the expression produced another integer and assignment changed the `attempts` binding.

4. Mutating methods usually return `None`

Several standard methods emphasize their effect by returning `None`:

```python
items = ["map", "rope"]
mapping = {"north": "open"}
visited = {(0, 0)}

assert items.sort() is None
assert mapping.update({"east": "closed"}) is None
assert visited.add((1, 0)) is None

assert items == ["map", "rope"]
assert mapping == {"north": "open", "east": "closed"}
assert visited == {(0, 0), (1, 0)}
```

The returned `None` prevents an easy confusion between the action and a new collection result. This mistake loses the useful binding:

```python
items = ["map"]
items = items.append("rope")

assert items is None
```

Repair it by keeping the command separate:

```python
items = ["map"]
items.append("rope")

assert items == ["map", "rope"]
```

Not every mutating method returns `None`; always check its contract. The pattern is common enough to predict cautiously, not to invent a universal rule.

Checkpoint: mutation and returned values

Mutation and rebinding checkpoint

Track both object identity and the operation’s returned value.

1. What happens to a list’s identity when `append` mutates it?

- **A.** It remains the same while the list state changes
- **B.** It becomes the appended item’s identity

- **C.** It changes to None
- **D.** It matches every equal list

2. Why must code keep the result of `mission_name.upper()` to use the uppercase value?

- **A.** Upper deletes the original name
- **B.** Strings are immutable, so the method returns a new value
- **C.** Upper mutates every alias
- **D.** Only lists can be assigned

3. Why does `items = items.append('rope')` make items become None?

- **A.** The list was empty
- **B.** Assignment freezes the list
- **C.** Append mutates the list but its returned value is None
- **D.** Rope is not hashable

4. What does `route = route + ['tower']` do?

- **A.** Mutates every nested object
- **B.** Deletes the old list
- **C.** Makes all aliases identical
- **D.** Creates a new list and rebinds route

Write your choices before checking the answer key at the end of this book.

5. Returning alternatives preserve the source

Compare methods and expressions that return new values:

```python
numbers = [3, 1, 2]
original_numbers = numbers

ordered = sorted(numbers)
assert ordered == [1, 2, 3]
assert numbers == [3, 1, 2]
assert ordered is not numbers
assert original_numbers is numbers

regions = {"north"}
expanded = regions | {"east"}
assert regions == {"north"}
assert expanded == {"north", "east"}
```

`sorted` accepts an iterable and returns a list. Set union with `|` returns a set. Their in-place counterparts have different ownership effects:

```python
numbers.sort()
regions.update({"east"})

assert numbers == [1, 2, 3]
assert regions == {"north", "east"}
```

Neither style is always preferable. Choose from whether the caller owns the source, whether aliases should observe the update, and whether preserving the old value is useful.

6. `+=` follows the left operand's type

With a list, augmented assignment normally performs in-place addition:

```python
items = ["map"]
alias = items

items += ["rope"]

assert items is alias
assert alias == ["map", "rope"]
```

With a tuple, there is no in-place tuple mutation. A new tuple is returned and the target name is rebound:

```python
steps = ("gate",)
old_steps = steps

steps += ("bridge",)

assert steps == ("gate", "bridge")
assert old_steps == ("gate",)
assert steps is not old_steps
```

Strings and numbers similarly produce new values. Do not classify syntax alone as mutating. Ask what the left operand's type promises for that operation.

7. An immutable tuple can point to a mutable list

```python
record = ("Nova", ["map"])
gear = record[1]

gear.append("rope")

assert record == ("Nova", ["map", "rope"])
```

The tuple still contains the same two references in the same positions. Its second reference reaches a list whose state changed. Tuple immutability prevents replacing a tuple slot; it does not freeze every object reachable from the tuple.

This distinction matters for copying and hashability. A nested mutable object can leak state and can make the containing tuple unhashable.

Checkpoint: `+=` and nested mutability

Type-dependent change checkpoint

Use the left operand's type and the complete nested graph.

1. What will an alias observe after `items += ['rope']` when items is a list?

- **A.** Only the old list

- **B.** The list mutated to include rope
- **C.** A tuple replacement
- **D.** An unhashable error

2. Why does an old tuple alias retain the earlier tuple after `steps += ('bridge',)`?

- **A.** Tuples reject all addition
- **B.** The alias is automatically copied
- **C.** Tuple addition creates a value and the target name is rebound
- **D.** Bridge is mutable

3. What can change in `('Nova', ['map'])`?

- **A.** The tuple can replace its first slot
- **B.** The tuple becomes a dictionary
- **C.** Its identity changes on append
- **D.** The nested list can mutate while the tuple retains the same reference

4. What should you inspect before deciding whether `+=` mutates?

- **A.** The left operand’s type and operation contract
- **B.** The variable name length
- **C.** Whether the line prints
- **D.** The current hash number

Write your choices before checking the answer key at the end of this book.

8. Hashability supports reliable lookup

Dictionaries and sets use a hash to locate candidates efficiently, then equality to confirm a matching key. A hashable object promises hash/equality behavior stable enough for its lifetime in the collection.

```python
room_names = {
    (0, 0): "entrance",
    (1, 0): "mirror hall",
}
visited = {(0, 0), (1, 0)}

candidate = tuple([1, 0])
assert candidate in visited
assert room_names[candidate] == "mirror hall"
```

`candidate` is a distinct tuple constructed through a different path. It compares equal to `(1, 0)` and has compatible hash behavior, so it finds the same set member and dictionary entry.

The core invariant is: objects that compare equal must produce the same hash while used as keys. Unequal objects may still have equal hashes; the collection checks equality to resolve such collisions.

9. Common key choices depend on the complete value

Value	Usually hashable?	Key/set-member result
string, integer, bytes	yes	commonly suitable
tuple of hashable elements	yes	commonly suitable
frozenset of hashable elements	yes	suitable when unordered grouping matters
list	no	mutable and rejected
dictionary	no	mutable and rejected
set	no	mutable and rejected
tuple containing a list	no	nested list makes the complete tuple unhashable

Observe supported values:

```python
markers = {
    "entrance",
    7,
    (1, 2),
    frozenset({"north", "east"}),
}

assert (1, 2) in markers
assert frozenset({"east", "north"}) in markers
```

These calls reveal unsupported boundaries when run separately:

```python
list_key = [1, 2]
tuple_with_list = ("route", [1, 2])

# hash(list_key)
# hash(tuple_with_list)
```

Each uncommented call raises `TypeError` because the complete value is not hashable. Unit 8 develops systematic exception handling; here the error directly answers the key-suitability question.

10. Hashes are not secure or persistent IDs

`hash(value)` returns an integer used by hash-based collections. Do not treat it as:

- encryption;
- a password hash;
- a collision-free fingerprint;
- an object’s identity;
- a database identifier; or
- a value guaranteed to remain the same across Python runs.

For example, Python commonly randomizes string hashes between processes. The program-level contract is that a live hashable key remains findable in its collection, not that its printed hash belongs in permanent data.

11. Choose a key from the meaning of the data

A room coordinate fits a tuple because its two positions form one fixed value:

```python
coordinate = (3, 5)
visited = {coordinate}
```

A collection of permissions might fit a frozenset when order does not matter:

```python
permissions = frozenset({"read", "map"})
access_levels = {permissions: "scout"}

assert access_levels[frozenset({"map", "read"})] == "scout"
```

Do not convert a list to a tuple only to silence an error if its meaning is supposed to change while stored. Redesign the key as a stable identifier, or keep mutable state in the dictionary value instead.

12. Lab: track an expedition state

Implement four focused operations:

```python
def add_supply_in_place(state, supply):
    """Append supply to state and return None."""
    raise NotImplementedError


def with_status(state, status):
    """Return a new outer state with status and shared untouched values."""
    raise NotImplementedError


def visit_room_in_place(state, coordinate):
    """Add one hashable coordinate to visited and return None."""
    raise NotImplementedError


def observation_counts(observations):
    """Return counts keyed by hashable observation labels."""
    raise NotImplementedError
```

Use this fixture and evidence:

```python
state = {
    "name": "Nova",
    "status": "searching",
    "supplies": ["water"],
    "visited": {(0, 0)},
}
alias = state

assert add_supply_in_place(state, "rope") is None
assert alias["supplies"] == ["water", "rope"]
```

PYTHON (CONTINUED)

```
updated = with_status(state, "ready")
assert updated == {
    "name": "Nova",
    "status": "ready",
    "supplies": ["water", "rope"],
    "visited": {(0, 0)},
}
assert updated is not state
assert state["status"] == "searching"

assert visit_room_in_place(state, (1, 0)) is None
assert state["visited"] == {(0, 0), (1, 0)}
assert observation_counts(["rune", "key", "rune"]) == {
    "rune": 2,
    "key": 1,
}
assert observation_counts([]) == {}
```

`with_status` changes only an outer immutable field, so a shallow outer copy is enough for this contract. Do not mutate `supplies` through `updated`; Lesson 3 will show why that different requirement needs a deeper ownership decision.

Hint: separate in-place commands from returned transformations

The two `_in_place` functions call mutating collection methods and rely on their implicit `None` result. `with_status` can use `state.copy()` then replace only the status slot. Count observations with a local dictionary whose string keys are hashable.

Show one complete solution after attempting the lab

PYTHON

```
def add_supply_in_place(state, supply):
    """Append supply to state and return None."""
    state["supplies"].append(supply)


def with_status(state, status):
    """Return a new outer state with status and shared untouched values."""
    updated = state.copy()
    updated["status"] = status
    return updated


def visit_room_in_place(state, coordinate):
    """Add one hashable coordinate to visited and return None."""
    state["visited"].add(coordinate)


def observation_counts(observations):
    """Return counts keyed by hashable observation labels."""
    counts = {}
    for observation in observations:
        counts[observation] = counts.get(observation, 0) + 1
    return counts
```

Checkpoint: hashable expedition state

Expedition state checkpoint

Match each operation to its mutation and lookup contract.

1. Why can `(1, 0)` be stored in the visited set?

- **A.** It is a tuple containing only hashable integers
- **B.** Every coordinate is global
- **C.** Sets accept all lists and tuples
- **D.** Its printed hash is permanent

2. Why does alias observe the added rope?

- **A.** with_status copied it
- **B.** alias and state reach the same outer dictionary and nested list
- **C.** String keys synchronize objects
- **D.** Sets force list mutation

3. Why is a shallow copy sufficient for with_status's stated contract?

- **A.** It freezes every nested object
- **B.** Status is a list
- **C.** Only the outer status slot is replaced; nested state is not edited
- **D.** Deepcopy cannot copy strings

4. Why would `[1, 0]` fail as a set member?

- **A.** It has two elements
- **B.** Coordinates must be strings
- **C.** Lists compare by identity only
- **D.** A mutable list is unhashable

Write your choices before checking the answer key at the end of this book.

13. Explain each update and lookup

Use your lab to answer:

1. Which functions mutate an object and which return a new outer object?
2. Why does a returned `None` not mean the in-place commands failed?
3. Which nested values remain shared by `with_status`, and why is that acceptable only under its narrow contract?
4. Why do equal tuple coordinates find one set entry?
5. Why would converting a changing route list into a key be a design problem even if a tuple conversion silenced `TypeError`?

Then change `with_status` so it also appends a supply. Predict the leak through the shallow copy before running. Revert the change; Lesson 3 will implement the appropriate deeper strategy.

Key points

Key points

- Mutation changes an existing object’s state; rebinding makes a name reach a different object.
- Common mutating collection methods return `None`; keep the command separate from the useful collection binding.
- Returning alternatives such as `sorted` preserve their source, while in-place counterparts make aliases observe a change.
- `+=` follows the left operand’s type: lists commonly mutate, while tuples, strings, and numbers produce values and rebind the target.
- An immutable container can still refer to mutable descendants.
- Hashability supports stable dictionary/set lookup and depends on the complete value, not merely tuple punctuation.
- A hash is not encryption, object identity, or a persistent record identifier. :::

References

- Python data model: object mutability and identity
- Python library reference: mutable and immutable sequence operations
- Python glossary: hashable
- Python built-in function: `hash`

Copying Nested Data Without Surprises

Online lesson and runnable notebook

Diagnose repeated references at every depth and choose reconstruction, shallow copying, selective copying, or deep copying from explicit ownership needs.

- **Level:** Python Foundations · Unit 6
- **Estimated time:** 3.5–5 hours
- **You will learn:** Trace nested aliases, recognize one-level copy operations, preserve intentional sharing, and choose selective or deep independence only where ownership requires it.
- **Practice in:** Google Colab, JupyterLab, or a local editor

1. Repeating one row repeats its reference

This looks like a three-row grid:

```python
row = [0, 0, 0]
grid = [row] * 3

grid[0][1] = 7

assert grid == [
    [0, 7, 0],
    [0, 7, 0],
    [0, 7, 0],
]
assert grid[0] is grid[1] is grid[2]
```

List repetition copied the reference to `row` three times. It did not evaluate a new row construction for every position. The cell assignment followed `grid[0]` to the one row object and mutated it. Every apparent row displays the same changed object.

This lesson answers:

- At which depth do two structures still share mutable state?
- What exactly does a shallow copy separate?
- When is copying selected nested fields clearer than `deepcopy`?
- Why can a deep-copied graph still contain intentional internal sharing?

2. Construct a fresh row for every position

A comprehension executes its expression once per iteration:

```python
grid = [[0, 0, 0] for _ in range(3)]

grid[0][1] = 7
```

PYTHON (CONTINUED)

```
assert grid == [
    [0, 7, 0],
    [0, 0, 0],
    [0, 0, 0],
]
assert grid[0] is not grid[1]
assert grid[1] is not grid[2]
```

The outer list now holds references to three independently constructed rows. This is reconstruction, not a later copy operation. When you control creation, constructing the intended graph directly is often clearer than creating aliases and repairing them afterward.

3. Assignment, construction, and copying answer different questions

PYTHON

```
source = {"name": "Nova", "tasks": ["map"]}
alias = source
constructed = {"name": source["name"], "tasks": ["map"]}

assert alias is source
assert constructed is not source
assert constructed["tasks"] is not source["tasks"]
```

- Assignment adds a path to an existing object.
- The explicit dictionary/list literals construct the objects written in the expression.
- A copy operation follows a defined copy depth and may retain inner references.

Do not infer nested independence merely because the outer object is new. Follow every path that later code will mutate.

4. A shallow copy separates the outside and shares the inside

PYTHON

```
template = {
    "name": "scout",
    "tasks": ["map gate"],
    "notes": {"entries": ["start"]},
}
shallow = template.copy()

assert shallow is not template
assert shallow["tasks"] is template["tasks"]
assert shallow["notes"] is template["notes"]
assert shallow["notes"]["entries"] is template["notes"]["entries"]
```

The new dictionary has its own set of key/value slots, but each value slot is initialized with a reference from the original. Replacing an outer slot remains local to one dictionary:

PYTHON

```
shallow["name"] = "navigator"

assert template["name"] == "scout"
assert shallow["name"] == "navigator"
```

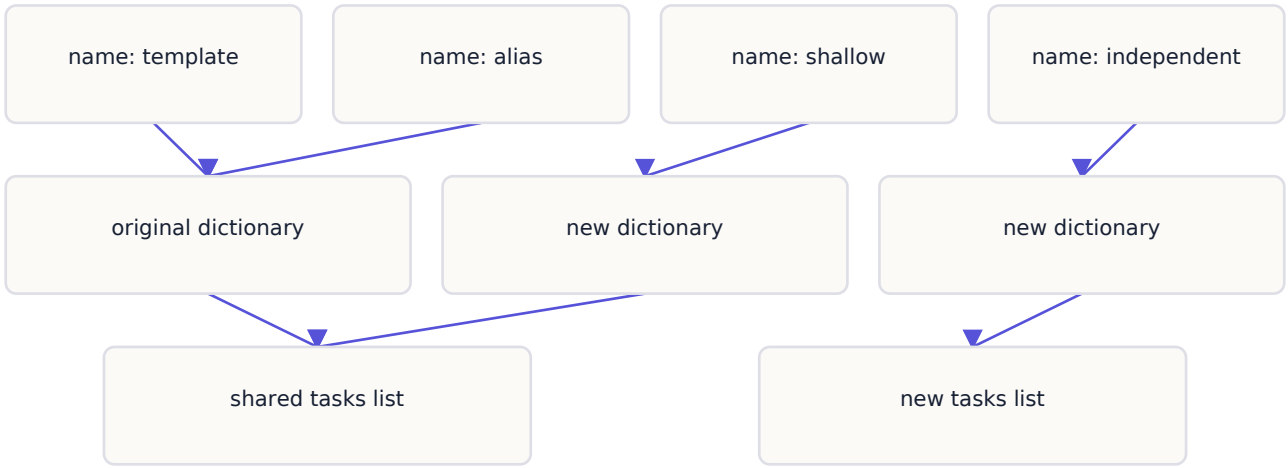
Mutating a shared descendant leaks:

```python
shallow["tasks"].append("open bridge")

assert template["tasks"] == ["map gate", "open bridge"]
```

The copy is neither “broken” nor “half complete.” It correctly implements a one-level copy. The problem is choosing that depth for a contract that required nested independence.

Assignment shares the outer dictionary. A shallow copy creates another outer dictionary, but its task slot still points to the original nested list.



Checkpoint: repeated and shallow references

Shallow object graph checkpoint

Follow the exact path to the object that changes.

1. Why does changing `grid[0][1]` affect every row in `[row] * 3`?

- **A.** Every outer position stores a reference to the same row
- **B.** Integers synchronize lists
- **C.** Item assignment deep-copies rows
- **D.** Range forces equality

2. Why does `[[0, 0] for _ in range(3)]` create independent rows?

- **A.** Underscore copies values
- **B.** The inner list expression runs once per iteration
- **C.** All comprehensions deep-copy
- **D.** Zero is mutable

3. What does `template.copy()` guarantee for a dictionary?

- **A.** Every nested value is independent

- **B.** All values become immutable
- **C.** A distinct outer dictionary is created
- **D.** The source is deleted

4. Why does appending through `shallow['tasks']` affect template?

- **A.** Names with matching keys are linked
- **B.** Dictionary equality mutates
- **C.** Strings propagate changes
- **D.** Both task slots reach the same nested list

Write your choices before checking the answer key at the end of this book.

5. Several familiar operations make shallow copies

For lists, these create distinct outer lists and reuse element references:

PYTHON
```
nested = [["map"], ["key"]]

by_method = nested.copy()
by_slice = nested[:]
by_constructor = list(nested)

assert by_method is not nested
assert by_slice is not nested
assert by_constructor is not nested
assert by_method[0] is nested[0]
assert by_slice[0] is nested[0]
assert by_constructor[0] is nested[0]
```

For dictionaries, `.copy()`, `dict(existing)`, unpacking, merge, and `copy.copy()` are one-level operations:

PYTHON
```
from copy import copy

source = {"tasks": ["map"]}
copies = [
    source.copy(),
    dict(source),
    {**source},
    source | {},
    copy(source),
]

assert all(candidate is not source for candidate in copies)
assert all(candidate["tasks"] is source["tasks"] for candidate in copies)
```

These spellings have different uses and type constraints, but none recursively copies nested objects. Do not choose among them expecting a deeper ownership boundary.

6. Immutable nested values can remain shared safely

```python
template = {
    "rules": ("keep the map", "share clues"),
    "difficulty": "foundation",
    "tasks": ["find gate"],
}
shallow = template.copy()

assert shallow["rules"] is template["rules"]
assert shallow["difficulty"] is template["difficulty"]
```

The identity of immutable literals may be reused, but the design does not depend on it. More importantly, neither tuple's slots nor string contents can be mutated. Sharing those particular values cannot leak an in-place content change.

The mutable `tasks` list is different. If each board may add tasks, that field must become independently owned. Copy decisions can therefore vary by field.

7. Selective copying follows an ownership table

Write the contract before code:

Field	May new owner mutate it?	Required relationship
`rules` tuple	no	sharing is safe
`difficulty` string	replaced only	sharing current value is safe
`tasks` list	yes	independent list
`notes` dictionary	yes	independent dictionary
`notes["entries"]` list	yes	independent list

Implement exactly that boundary:

```python
def copy_board(template):
    board = template.copy()
    board["tasks"] = template["tasks"].copy()
    board["notes"] = template["notes"].copy()
    board["notes"]["entries"] = template["notes"]["entries"].copy()
    return board
```

Selective copying makes the intended ownership visible. It requires maintenance when the data shape changes, which can be an advantage: adding a mutable field forces the designer to decide its ownership instead of silently copying everything.

Checkpoint: ownership and selective copying

Selective ownership checkpoint

Copy the paths that future code may mutate independently.

1. Why may two boards share the immutable rules tuple under this contract?

- **A.** Tuples deep-copy every child
- **B.** Neither board can mutate the tuple’s fixed immutable rule values
- **C.** Identity makes rules equal
- **D.** Dictionaries reject tuples

2. Why copy both the notes dictionary and its entries list?

- **A.** Dictionary copies remove strings
- **B.** Lists require global state
- **C.** Both levels may be mutated independently by the new owner
- **D.** Deepcopy cannot handle dictionaries

3. What determines the minimum necessary copy depth?

- **A.** The number of printed lines
- **B.** Whether keys are alphabetic
- **C.** The largest available library function
- **D.** Which mutable descendants must have independent future changes

4. What should happen when a new mutable field is added to a selectively copied record?

- **A.** Its ownership should be decided and the copy function updated if needed
- **B.** It automatically becomes immutable
- **C.** It must always remain shared
- **D.** The whole program must use JSON

Write your choices before checking the answer key at the end of this book.

8. `deepcopy` creates an independent supported graph

PYTHON

```
from copy import deepcopy

template = {
    "name": "scout",
    "tasks": ["map gate"],
    "notes": {"entries": ["start"]},
}
independent = deepcopy(template)

assert independent is not template
assert independent["tasks"] is not template["tasks"]
assert independent["notes"] is not template["notes"]
assert independent["notes"]["entries"] is not template["notes"]["entries"]

independent["notes"]["entries"].append("bridge found")
assert template["notes"]["entries"] == ["start"]
```

`deepcopy` recursively copies supported mutable descendants and reuses immutable values when appropriate. It is concise when the complete mutable graph truly needs independent edits.

9. Deep copy preserves internal sharing in the new graph

Suppose two source paths intentionally refer to one list:

```python
from copy import deepcopy

shared_clues = ["moon"]
source = {
    "visible_clues": shared_clues,
    "all_clues": shared_clues,
}

clone = deepcopy(source)

assert clone is not source
assert clone["visible_clues"] is not source["visible_clues"]
assert clone["visible_clues"] is clone["all_clues"]
```

The copied graph has a new clues list separated from the source, but both clone paths still share that one new list. `deepcopy` remembers objects it has already copied so it can preserve relationships and handle recursive container graphs. It does not blindly duplicate one object every time another path encounters it.

That behavior is usually desirable: graph structure is part of the data. If the two clone fields must become independent despite being shared in the source, that is a transformation requiring explicit reconstruction, not ordinary deep copying.

10. Deep copy is not a universal safety button

Deep copying can:

- consume significant time and memory for large graphs;
- duplicate mutable objects that should remain intentionally shared;
- preserve internal aliases that the new contract wanted to split;
- interact with custom class copy hooks introduced later; and
- be unsuitable or return unchanged references for resource-like or executable objects such as modules, functions, open files, sockets, and external sessions.

Do not use JSON encode/decode as a generic substitute. Serialization accepts a limited data model and can change tuples, sets, keys, bytes, custom objects, and other semantics. Unit 9 teaches external formats for their actual purpose.

11. Choose the strategy from the required future edits

Strategy	Choose it when	Evidence to record
intentional sharing	all owners should observe updates	relevant paths are identical
reconstruct	source data can clearly build the intended graph	each constructed mutable owner is distinct
shallow copy	only outer membership/fields change	outer identity differs; nested identities may match
selective copy	known descendants need separate ownership	specified mutable paths differ
deep copy	a supported complete graph needs independence	source and copied mutable descendants differ

Avoid “copy just in case.” State the future mutation and source-preservation promise first. The assertions then describe the minimum graph relationship that can satisfy it.

12. Lab: create independent mission boards

Implement a shallow comparison and a selectively owned board:

```python
def shallow_board(template):
    """Return a new outer dictionary that shares nested values."""
    raise NotImplementedError


def mission_board(template, *, owner):
    """Return a board whose mutable task and note state belongs to owner."""
    raise NotImplementedError


def board_relationship(left, right):
    """Return identity evidence for important board paths."""
    raise NotImplementedError
```

Use this template:

```python
template = {
    "owner": None,
    "difficulty": "foundation",
    "rules": ("record evidence", "preserve source"),
    "tasks": {
        "north": ["map gate"],
        "south": [],
    },
    "notes": {"entries": ["start"]},
}

shallow = shallow_board(template)
assert shallow is not template
assert shallow["tasks"] is template["tasks"]

nova = mission_board(template, owner="Nova")
sol = mission_board(template, owner="Sol")
```

PYTHON (CONTINUED)

```
assert nova["owner"] == "Nova"
assert sol["owner"] == "Sol"
assert nova["rules"] == template["rules"]
assert nova["tasks"] is not template["tasks"]
assert nova["tasks"]["north"] is not template["tasks"]["north"]
assert nova["notes"] is not template["notes"]
assert nova["notes"]["entries"] is not template["notes"]["entries"]

nova["tasks"]["north"].append("open bridge")
nova["notes"]["entries"].append("rune found")

assert template["tasks"]["north"] == ["map gate"]
assert template["notes"]["entries"] == ["start"]
assert sol["tasks"]["north"] == ["map gate"]
assert sol["notes"]["entries"] == ["start"]
assert board_relationship(nova, sol) == {
    "same_outer": False,
    "same_tasks": False,
    "same_north_tasks": False,
    "same_notes": False,
    "same_note_entries": False,
}
```

Keep the immutable `rules` tuple available for safe sharing. The lab does not require that its identity be shared because equivalent construction or implementation choices can differ; its value and immutability satisfy the contract.

Hint: rebuild every mutable level named in the ownership table

Begin with a shallow outer copy. Replace `tasks` with a new dictionary whose values are new lists. Replace `notes` with a new dictionary and replace its `entries` with a new list. Rebind `owner`. The relationship helper uses `is` at each listed path.

Show one complete solution after attempting the lab

PYTHON

```
def shallow_board(template):
    """Return a new outer dictionary that shares nested values."""
    return template.copy()


def mission_board(template, *, owner):
    """Return a board whose mutable task and note state belongs to owner."""
    board = template.copy()
    board["owner"] = owner
    board["tasks"] = {}
    for direction, tasks in template["tasks"].items():
        board["tasks"][direction] = tasks.copy()
    board["notes"] = template["notes"].copy()
    board["notes"]["entries"] = template["notes"]["entries"].copy()
    return board


def board_relationship(left, right):
    """Return identity evidence for important board paths."""
    return {
        "same_outer": left is right,
        "same_tasks": left["tasks"] is right["tasks"],
        "same_north_tasks": left["tasks"]["north"] is right["tasks"]["north"],
        "same_notes": left["notes"] is right["notes"],
        "same_note_entries": (
            left["notes"]["entries"] is right["notes"]["entries"]
        ),
    }
```

Checkpoint: mission-board independence

Mission board checkpoint

Check the identity of each mutable ownership boundary.

1. Why must mission_board create both a new tasks dictionary and new task lists?

- **A.** Only strings may be shared
- **B.** The owner may add directions and edit each direction’s tasks independently
- **C.** Dictionaries automatically deep-copy lists
- **D.** Tuple rules require it

2. What proves Nova’s new task did not leak into Sol’s board?

- **A.** Their owner strings differ
- **B.** Their rules compare equal
- **C.** Sol’s north tasks retain only `map gate`
- **D.** Nova’s outer dictionary is truthy

3. Which evidence identifies the shallow board relationship?

- **A.** The outer dictionaries are identical
- **B.** Every nested value is copied
- **C.** The source is empty
- **D.** The outer dictionaries differ while their tasks dictionaries are identical

4. When would deepcopy be reasonable instead of the selective lab implementation?

- **A.** When the supported complete mutable graph needs independent ownership
- **B.** Whenever a string field changes
- **C.** To make hashes secure
- **D.** To rebind one local name

Write your choices before checking the answer key at the end of this book.

13. Defend the selected copy depth

Use your lab to explain:

1. Which outer and nested objects are new for Nova?
2. Which values may remain shared safely, and why?
3. Why would copying only the tasks dictionary still leak list edits?
4. What schema change would require updating `mission_board`?
5. What would `deepcopy(template)` make easier, and what ownership decisions would it make less visible?

Add an `evidence` list inside `notes`. Predict the leak before updating the copy function, observe it once, then give the new field independent ownership and rerun every source-preservation assertion.

Key points

Key points

- Repetition can duplicate references rather than nested objects; construct a fresh mutable value for each intended owner.
- A shallow copy creates a new outer container and reuses its contained references.
- Immutable nested values can often remain shared safely.
- Selective copying makes field-level ownership decisions explicit.
- `deepcopy` separates a supported object graph from its source while preserving intentional sharing relationships inside the cloned graph.
- Deep copy has time, memory, resource, and semantic tradeoffs; it is not a universal safety button.
- Choose copy depth from the exact future mutations that must remain independent. :::

References

- Python standard library: shallow and deep copy operations
- Python FAQ: multidimensional list construction
- Python tutorial: copying lists
- Python language reference: display expressions and container construction

Designing Functions That Control Mutation

Online lesson and runnable notebook

Trace object sharing across calls and design explicit contracts for in-place effects, returned copies, mutable defaults, retained inputs, and safe snapshots.

- **Level:** Python Foundations · Unit 6
- **Estimated time:** 3.5–5 hours
- **You will learn:** Distinguish local parameter rebinding from caller-visible mutation, design in-place and copying APIs, prevent shared defaults, and protect retained or returned mutable state.
- **Practice in:** Google Colab, JupyterLab, or a local editor

1. A helper can change caller-owned state

PYTHON

```
def record_badge(player, badge):
    player["badges"].append(badge)


nova = {"name": "Nova", "badges": ["starter"]}
result = record_badge(nova, "vault scout")

assert result is None
assert nova["badges"] == ["starter", "vault scout"]
```

The parameter name `player` is local to the call, but it reaches the dictionary supplied by the caller. The nested append mutates caller-owned state. If the caller expected a separate returned record, the function’s ownership contract is wrong even though the Python is valid.

This lesson answers:

- Why can mutation cross a call boundary while parameter rebinding does not?
- How should an API distinguish an in-place command from a transformer?
- When should a function copy an input or returned result defensively?
- How do mutable defaults and returned aliases leak state between calls?

2. Argument evaluation creates a local binding to the object

When Python evaluates `record_badge(nova, "vault scout")`:

1. the caller evaluates `nova` and obtains its dictionary object;
2. the function call binds local parameter `player` to that object;
3. the caller name and local parameter temporarily reach the same dictionary;
4. mutation through either path changes that one object; and

5 the local parameter binding disappears when the call returns, but the changed dictionary remains reachable through `nova`.

This is sometimes called “call by sharing” or “pass by object reference.” Avoid saying only “pass by reference,” which can wrongly suggest that assigning the parameter will assign the caller’s variable.

The caller and parameter reach one object during an in-place call. A copying transformer instead creates a new object and returns its reference.



local parameter: player → new dictionary: copying transformer creates

new dictionary → caller name: updated: return reference

3. Rebinding a parameter cannot rebind the caller’s name

PYTHON
```
def replace_route(route):
    route = ["new gate"]
    return route


original = ["old gate"]
replacement = replace_route(original)

assert original == ["old gate"]
assert replacement == ["new gate"]
assert replacement is not original
```

Assignment changes the local `route` binding. The caller name `original` is in a different namespace and still reaches its old list. Returning the new object lets the caller decide whether to bind another name to it.

Compare mutation:

PYTHON
```
def replace_first_in_place(route):
    route[0] = "new gate"


original = ["old gate"]
replace_first_in_place(original)
```

PYTHON (CONTINUED)

```
assert original == ["new gate"]
```

Item assignment follows the parameter reference and changes the shared list object; it does not rebind the parameter.

4. Put the ownership choice in the public contract

Use two functions when callers need two different policies:

PYTHON

```
def add_badge_in_place(player, badge):
    """Append badge to caller-owned player; return None."""
    player["badges"].append(badge)


def with_badge(player, badge):
    """Return an independent player containing badge; preserve player."""
    from copy import deepcopy

    updated = deepcopy(player)
    updated["badges"].append(badge)
    return updated
```

Evidence distinguishes them:

PYTHON

```
source = {"name": "Nova", "badges": ["starter"]}

updated = with_badge(source, "vault scout")
assert source["badges"] == ["starter"]
assert updated["badges"] == ["starter", "vault scout"]
assert updated["badges"] is not source["badges"]

result = add_badge_in_place(source, "pathfinder")
assert result is None
assert source["badges"] == ["starter", "pathfinder"]
```

The suffix `_in_place` is not required Python syntax, but it communicates an effect. Names such as `update`, `record`, or `append` can also signal mutation when the surrounding API and docstring make ownership clear.

Checkpoint: rebinding and caller-visible effects

Call boundary checkpoint

Follow the caller name, local parameter, and object separately.

1. What does the parameter `player` initially refer to in `record_badge(nova, badge)`?

- **A.** The same dictionary object produced by evaluating nova
- **B.** An automatic deep copy
- **C.** The caller’s name object
- **D.** A tuple containing every argument

2. Why does `route = ['new gate']` inside a function not rebind the caller’s original name?

- **A.** Lists cannot be returned

- **B.** It changes only the function’s local parameter binding
- **C.** Assignment always mutates
- **D.** The caller’s list is immutable

3. Why is `route[0] = 'new gate'` visible to the caller?

- **A.** The parameter becomes global
- **B.** Return copies the change
- **C.** Item assignment mutates the shared list object
- **D.** Strings rebind the caller

4. Which contract most clearly announces a caller-visible effect?

- **A.** A helper with no name
- **B.** A function that silently changes nested state
- **C.** A transformer documented as source-preserving but mutating
- **D.** `add_badge_in_place` documented to append and return None

Write your choices before checking the answer key at the end of this book.

5. Standard APIs offer both effect and result styles

Python already uses these contrasting contracts:

```python
numbers = [3, 1, 2]
alias = numbers

ordered = sorted(numbers)
assert ordered == [1, 2, 3]
assert numbers == [3, 1, 2]

result = numbers.sort()
assert result is None
assert alias == [1, 2, 3]
```

Other pairs include:

In-place effect	Returned alternative
`set.update(other)`	`left | right`
`dict.update(other)`	`left | right` on supported Python versions
`list.reverse()`	`list(reversed(values))`
item assignment	a reconstruction or explicit copy-and-edit helper

The returned alternative is not necessarily a deep independent result. A new outer list can still contain the same nested objects. Copy depth remains part of the contract.

6. Mutating and returning the same object can hide aliases

```python
def add_and_return(items, item):
    items.append(item)
    return items


source = ["map"]
result = add_and_return(source, "rope")

assert result is source
```

This API permits chaining but may look like a transformer that produced a new list. It is valid if documented, yet callers must understand that `result` is a new alias. Many built-in in-place methods return `None` precisely to make the effect style harder to mistake for independent output.

Choose one clear primary story. If a function mutates and returns the same object, state both facts and test identity explicitly.

7. A mutable default belongs to the definition, not each call

```python
def remember_badge(badge, history=[]):
    history.append(badge)
    return history


first = remember_badge("starter")
second = remember_badge("vault scout")

assert first is second
assert second == ["starter", "vault scout"]
```

The list expression runs once when Python executes the `def` statement. Every call that omits `history` binds its local parameter to that same default list. This is a definition-time ownership decision, not random memory.

Create per-call state explicitly:

```python
def remember_badge(badge, history=None):
    if history is None:
        history = []
    history.append(badge)
    return history


first = remember_badge("starter")
second = remember_badge("vault scout")

assert first == ["starter"]
assert second == ["vault scout"]
assert first is not second
```

When a caller supplies a list, this version intentionally mutates it:

```python
shared_history = []
result = remember_badge("starter", shared_history)
```

PYTHON (CONTINUED)

```
assert result is shared_history
assert shared_history == ["starter"]
```

The default case and explicit-input case can have different ownership, so the docstring should state that supplied history is updated.

Checkpoint: effect style and defaults

Mutation contract checkpoint

Identify who owns the changed object before and after each call.

1. Which call returns an ordered list while preserving the source outer list?

- **A.** `items.sort()`
- **B.** `sorted(items)`
- **C.** `items.append(1)`
- **D.** `items.reverse()`

2. What does `result is source` prove after add_and_return?

- **A.** The values are merely equal
- **B.** The source was deep-copied
- **C.** The result is another reference to the mutated source object
- **D.** The function had no side effect

3. When is the list in `history=[]` created?

- **A.** After every append
- **B.** Whenever the result is printed
- **C.** Only after a failure
- **D.** Once when Python executes the function definition

4. How does the None-sentinel repair create independent omitted histories?

- **A.** The body constructs a new list during each call that receives None
- **B.** None freezes the original list
- **C.** The caller’s name is deleted
- **D.** Every list becomes a tuple

Write your choices before checking the answer key at the end of this book.

8. Copying on input protects retained state

A function or closure may retain an input after the call. Storing the caller’s list directly shares future changes:

```python
def make_route_reader(route):
    def read_route():
        return route

    return read_route


source = ["gate"]
read_route = make_route_reader(source)
source.append("bridge")

assert read_route() == ["gate", "bridge"]
```

If the contract says the reader captures a snapshot at creation, copy on input:

```python
def make_route_reader(route):
    saved_route = list(route)

    def read_route():
        return saved_route

    return read_route


source = ["gate"]
read_route = make_route_reader(source)
source.append("bridge")

assert read_route() == ["gate"]
```

This shallow copy is enough only because the route contains immutable strings. A nested mutable route would require a deeper ownership decision.

9. Returning internal mutable state creates another leak

The copied-input version still returns its internal list directly:

```python
source = ["gate"]
read_route = make_route_reader(source)
leaked = read_route()
leaked.append("secret door")

assert read_route() == ["gate", "secret door"]
```

The caller received an alias to retained state. Return an immutable snapshot or a suitable copy:

```python
def make_safe_route_reader(route):
    saved_route = list(route)

    def read_route():
        return tuple(saved_route)

    return read_route


source = ["gate"]
read_route = make_safe_route_reader(source)
snapshot = read_route()
```

PYTHON (CONTINUED)

```
assert snapshot == ("gate",)
assert source == ["gate"]
```

The tuple works because its elements are immutable strings and the caller needs a read-only sequence. A dictionary or nested mutable snapshot may need explicit copying instead.

10. Defensive copying has a cost

Copying at every boundary can consume time and memory, hide useful intentional sharing, and create stale snapshots. Not copying can expose state to changes from places that did not appear in the public call.

For every retained input or returned collection, state:

1. Who created this object?
2. Who is allowed to mutate it?
3. Will the function retain it after returning?
4. Does the result share mutable descendants with the input or internal state?
5. Which source objects must remain unchanged?

Then choose no copy, a shallow/selective copy, a deep copy, or an immutable representation. This is an API decision, not a style requirement.

11. Value checks and identity checks prove different promises

PYTHON

```
source = {"badges": ["starter"]}
updated = with_badge(source, "vault scout")

assert source == {"badges": ["starter"]}
assert updated == {"badges": ["starter", "vault scout"]}
assert updated is not source
assert updated["badges"] is not source["badges"]
```

The first two assertions establish values. The final two establish ownership relationships required for safe later edits. Do not add identity assertions for immutable fields unless identity itself belongs to the contract.

12. Lab: publish a mutation-aware quest API

Implement four ownership contracts:

PYTHON

```
def record_clue_in_place(state, room, clue):
    """Record room and clue in caller-owned state; return None."""
    raise NotImplementedError


def with_clue(state, room, clue):
    """Return independent state containing room and clue; preserve state."""
    raise NotImplementedError
```

PYTHON (CONTINUED)

```
def remember_event(event, history=None):
    """Append event to supplied history or a fresh per-call list; return it."""
    raise NotImplementedError


def journal_snapshot(state):
    """Return an immutable snapshot of rooms and clues."""
    raise NotImplementedError
```

Use this state and evidence:

PYTHON

```
source = {
    "name": "Nova",
    "journal": {
        "rooms": [(0, 0)],
        "clues": ["entrance"],
    },
}

assert record_clue_in_place(source, (1, 0), "mirror") is None
assert source["journal"] == {
    "rooms": [(0, 0), (1, 0)],
    "clues": ["entrance", "mirror"],
}

before = {
    "name": "Nova",
    "journal": {
        "rooms": [(0, 0), (1, 0)],
        "clues": ["entrance", "mirror"],
    },
}
updated = with_clue(source, (2, 0), "moon")

assert source == before
assert updated["journal"]["rooms"] == [(0, 0), (1, 0), (2, 0)]
assert updated["journal"]["clues"] == ["entrance", "mirror", "moon"]
assert updated is not source
assert updated["journal"] is not source["journal"]
assert updated["journal"]["rooms"] is not source["journal"]["rooms"]
assert updated["journal"]["clues"] is not source["journal"]["clues"]

first_history = remember_event("gate")
second_history = remember_event("bridge")
assert first_history == ["gate"]
assert second_history == ["bridge"]
assert first_history is not second_history

supplied_history = []
assert remember_event("tower", supplied_history) is supplied_history
assert supplied_history == ["tower"]

snapshot = journal_snapshot(updated)
assert snapshot == {
    "rooms": ((0, 0), (1, 0), (2, 0)),
    "clues": ("entrance", "mirror", "moon"),
}
assert isinstance(snapshot["rooms"], tuple)
assert isinstance(snapshot["clues"], tuple)
```

`with_clue` may use `deepcopy` for this small supported graph or explicit selective copying. Be prepared to explain the choice. `journal_snapshot` returns a new dictionary whose sequence values are immutable tuples.

Hint: make ownership visible in each function body

The in-place function appends directly. The transformer copies before reusing the in-place operation. The default helper constructs a list only when history is `None`. The snapshot converts each internal list to a tuple rather than returning the internal lists.

Show one complete solution after attempting the lab

PYTHON

```
from copy import deepcopy


def record_clue_in_place(state, room, clue):
    """Record room and clue in caller-owned state; return None."""
    state["journal"]["rooms"].append(room)
    state["journal"]["clues"].append(clue)


def with_clue(state, room, clue):
    """Return independent state containing room and clue; preserve state."""
    updated = deepcopy(state)
    record_clue_in_place(updated, room, clue)
    return updated


def remember_event(event, history=None):
    """Append event to supplied history or a fresh per-call list; return it."""
    if history is None:
        history = []
    history.append(event)
    return history


def journal_snapshot(state):
    """Return an immutable snapshot of rooms and clues."""
    return {
        "rooms": tuple(state["journal"]["rooms"]),
        "clues": tuple(state["journal"]["clues"]),
    }
```

Checkpoint: safe API boundaries

Quest API checkpoint

Identify the ownership promise proved by each assertion group.

1. Why can with_clue safely call the in-place helper on its copied state?

- **A.** The copy owns independent mutable descendants
- **B.** In-place calls never mutate
- **C.** Return automatically restores source
- **D.** Tuple rooms deep-copy dictionaries

2. What proves omitted-history calls do not share a default list?

- **A.** Both contain strings
- **B.** Their result lists are distinct and contain only their own event

- **C.** The function has a docstring
- **D.** Supplied history is empty

3. Why does the snapshot return tuples for rooms and clues?

- **A.** Tuples are always faster
- **B.** Lists cannot contain strings
- **C.** Callers cannot append through the snapshot into the retained lists
- **D.** Dictionary results require tuples

4. What happens when a caller explicitly supplies history to remember_event?

- **A.** A deep copy is always returned
- **B.** The argument is ignored
- **C.** The list becomes immutable
- **D.** The supplied list is intentionally updated and returned

Write your choices before checking the answer key at the end of this book.

13. Explain the ownership contract aloud

For each lab function, state:

1. who owns each mutable input before the call;
2. whether the function mutates, retains, or copies it;
3. which result is returned and whether it aliases anything;
4. which future caller mutation can cross the boundary; and
5. which assertions provide value and identity evidence.

Then deliberately return `state["journal"]["clues"]` from the snapshot, append through the result, and observe the leak. Restore the tuple snapshot and rerun from a clean state.

Key points

Key points

- Function arguments bind local parameter names to evaluated objects; Python does not automatically copy them.
- Rebinding a parameter stays local, while mutating an object through it is visible through caller references.
- In-place commands and source-preserving transformers should advertise distinct ownership contracts.
- Returning a mutated input creates another alias unless the API says otherwise.
- Mutable defaults are definition-owned objects reused across omitted-argument calls; construct per-call state with a sentinel.

- Copying retained input protects against later caller edits; copying or freezing output protects internal state from callers.
- Defensive copying costs time and memory, so apply it where ownership requires isolation rather than at every boundary. :::

References

- Python tutorial: default argument evaluation
- Python FAQ: changing a list inside a function
- Python standard library: shallow and deep copy operations
- Python tutorial: list methods and mutating return conventions

Unit Challenge: Break the Mirror Vault

Online lesson and runnable notebook

Repair a magical cloning spell with independent nested state, hashable room markers, deliberate in-place and returned-copy APIs, and safe snapshots.

- **Level:** Python Foundations · Unit 6 challenge
- **Estimated time:** 60–120 minutes
- **Outcome assessed:** Trace and repair nested aliases, choose stable room keys, and compose explicit mutation, copy, default, and snapshot contracts.
- **Evidence:** Pass 24 progressive assertions, preserve one debugging record, and explain the derived escape artifact.

1. Break a cloning spell without hard-coding the escape

The Mirror Vault creates adventurer doppelgängers from one blueprint. Its broken spell copies only names: explorers share inventories, journals, routes, and visited-room state. When Nova collects a rune, Sol’s journal changes too.

Build a small ownership-aware state API that produces independently editable explorers while safely sharing immutable vault rules. Nova must collect three runes and derive this final artifact:

```text
NOVA BREAKS THE MIRROR VAULT WITH 3 RUNES
```

Your implementation must make these behaviors visible:

1. spawning creates new outer and nested mutable owners;
2. tuple room coordinates work as stable set members;
3. one command mutates a named explorer intentionally;
4. one transformer returns independent state and preserves its source;
5. a snapshot does not expose internal lists;
6. omitted history gets fresh per-call storage; and
7. a shallow-copy variation reproduces the curse and is then repaired.

Keep the complete solution closed on your first attempt

Work for one or two focused hours. Run one assertion group at a time. Before opening a hint, identify the exact access path whose identity or value differs from the contract. :::

2. Read the blueprint and ownership rules

The blueprint contains:

- immutable role and difficulty strings;

- an immutable tuple of vault rules that may be shared safely;
- a mutable inventory list;
- a mutable journal dictionary containing rune and route lists; and
- a mutable set of visited tuple coordinates.

Constraints

1. Preserve all supplied public names, signatures, and docstrings.
2. Do not mutate `blueprint` while spawning or operating on an explorer.
3. Give each spawned explorer independent ownership of every mutable descendant.
4. Preserve the identity of the immutable `rules` tuple deliberately.
5. Use tuple coordinates such as `(1, 0)` as room markers; do not convert printed hashes into stored identifiers.
6. `record_rune_in_place` must mutate its supplied explorer and return `None`.
7. `with_supply` must return independently editable state and preserve its source.
8. `journal_snapshot` must not return either internal list directly.
9. `remember_attempt` must not share an omitted mutable default between calls.
10. Derive the final message from explorer state; do not assign the solved sentence directly in `vault_message`.
11. Keep calculation functions free of printing.
12. Use only the standard library. Classes, files, serialization, and exception handlers are outside this challenge.

3. Start from the contract

Run the blueprint unchanged:

PYTHON

```
blueprint = {
    "name": None,
    "role": "vault scout",
    "difficulty": "foundation",
    "rules": (
        "record evidence",
        "preserve the source",
        "trust tuple coordinates",
    ),
    "inventory": ["lantern", "rope"],
    "journal": {
        "runes": [],
        "route": [],
    },
    "visited": {(0, 0)},
}
```

Then copy this scaffold. Implement functions in the order shown:

```python
def spawn_explorer(blueprint, *, name):
    """Return an explorer with independent mutable state and shared rules."""
    raise NotImplementedError


def identity_report(left, right):
    """Return labeled identity evidence for important explorer paths."""
    raise NotImplementedError


def record_rune_in_place(explorer, room, rune):
    """Record room and rune in caller-owned explorer; return None."""
    raise NotImplementedError


def with_supply(explorer, supply):
    """Return independent explorer state containing one additional supply."""
    raise NotImplementedError


def journal_snapshot(explorer):
    """Return an immutable snapshot of rune and route sequences."""
    raise NotImplementedError


def remember_attempt(attempt, history=None):
    """Append attempt to supplied history or a fresh per-call list; return it."""
    raise NotImplementedError


def vault_message(explorer):
    """Return the explorer's derived Mirror Vault escape message."""
    raise NotImplementedError
```

`identity_report(left, right)` returns exactly these keys:

```python
{
    "same_outer": False,
    "same_inventory": False,
    "same_journal": False,
    "same_runes": False,
    "same_route": False,
    "same_visited": False,
    "same_rules": True,
}
```

The values above describe two correctly spawned explorers from the same blueprint.

4. Build the vault state in observable stages

1. Write an ownership table for every blueprint field. Mark immutable values as safe to share and mutable descendants as independently owned.
2. Implement `spawn_explorer` with selective copying. Check the outer dictionary, then every nested mutable path before performing a mutation.
3. Implement `identity_report` with `is` comparisons only. It reports relationships; it does not decide whether current values are correct.
4. Implement the in-place rune command. Add the tuple room to `visited`, append it to the journal route, append the rune, and rely on the implicit `None` result.

5 Implement `with_supply` by creating independent state first and then changing only the new inventory.

6 Convert journal lists to tuples for the returned snapshot.

7 Use `None` to create fresh history on omitted-argument calls while documenting that an explicitly supplied history is updated.

8 Count the explorer's runes inside `vault_message`; do not store the final count or sentence separately.

9 Run the shallow-copy variation last so its deliberate leak cannot contaminate the main blueprint.

Draw the path `nova -> journal -> runes` and the matching Sol and blueprint paths. The three final list objects must be distinct even when their initial contents are equal.

5. Run progressive assertions

Run these 24 numbered assertions unchanged. A failed identity assertion points to an ownership boundary; a failed value assertion points to an operation or result contract.

Spawn and identity evidence

PYTHON

```
from copy import deepcopy

before = deepcopy(blueprint)
nova = spawn_explorer(blueprint, name="Nova")
sol = spawn_explorer(blueprint, name="Sol")

assert nova["name"] == "Nova"                                    # 1
assert sol["name"] == "Sol"                                      # 2
assert blueprint == before                                        # 3
assert identity_report(nova, sol) == {                            # 4
    "same_outer": False,
    "same_inventory": False,
    "same_journal": False,
    "same_runes": False,
    "same_route": False,
    "same_visited": False,
    "same_rules": True,
}
assert nova["rules"] is blueprint["rules"]                        # 5
```

Deliberate mutation and hashable room evidence

PYTHON

```
result = record_rune_in_place(nova, (1, 0), "SUN")

assert result is None                                              # 6
assert nova["journal"]["runes"] == ["SUN"]                       # 7
assert nova["journal"]["route"] == [(1, 0)]                       # 8
assert (1, 0) in nova["visited"]                                  # 9
assert sol["journal"]["runes"] == []                             # 10
assert blueprint["journal"]["runes"] == []                       # 11
```

A list coordinate such as `[1, 0]` is unhashable and cannot be a set member. Keep this direct failure commented during the normal run:

PYTHON

```
# [1, 0] in nova["visited"]
```

Returned-copy and snapshot evidence

```python
powered = with_supply(nova, "mirror lens")

assert powered["inventory"] == ["lantern", "rope", "mirror lens"]     # 12
assert powered is not nova                                             # 13
assert powered["inventory"] is not nova["inventory"]                   # 14
assert powered["journal"] is not nova["journal"]                       # 15
assert nova["inventory"] == ["lantern", "rope"]                        # 16

snapshot = journal_snapshot(nova)
assert snapshot == {"runes": ("SUN",), "route": ((1, 0),)}           # 17
assert isinstance(snapshot["runes"], tuple) and isinstance(            # 18
    snapshot["route"], tuple
)
```

Fresh default and final artifact evidence

```python
first_attempt = remember_attempt("left mirror")
second_attempt = remember_attempt("right mirror")

assert first_attempt == ["left mirror"]                                # 19
assert second_attempt == ["right mirror"]                              # 20
assert first_attempt is not second_attempt                             # 21

record_rune_in_place(nova, (1, 1), "MOON")
record_rune_in_place(nova, (2, 1), "STAR")

artifact = vault_message(nova)
assert (                                                               # 22
    artifact == "NOVA BREAKS THE MIRROR VAULT WITH 3 RUNES"
    and blueprint == before
    and sol["journal"]["runes"] == []
)
```

Reproduce and repair the shallow-copy curse

Use a separate fixture so deliberate failure evidence does not damage the main blueprint:

```python
leaky_source = deepcopy(blueprint)
leaky_clone = leaky_source.copy()
leaky_clone["journal"]["runes"].append("SHADOW")

assert leaky_source["journal"]["runes"] == ["SHADOW"]                  # 23

repaired = spawn_explorer(leaky_source, name="Echo")
record_rune_in_place(repaired, (9, 9), "LIGHT")
assert (                                                               # 24
    leaky_source["journal"]["runes"] == ["SHADOW"]
    and repaired["journal"]["runes"] == ["SHADOW", "LIGHT"]
    and repaired["journal"]["runes"] is not leaky_source["journal"]["runes"]
)
```

Do not repair a leak by weakening identity evidence

Two lists can contain equal values and still need independent future ownership. Keep both value and identity assertions when the contract promises isolated mutation. :::

6. Use the hint ladder only when needed

Hint 1: mark the intended object graph

The new outer dictionary, inventory list, journal dictionary, rune list, route list, and visited set need new identities for every explorer. Immutable strings may be reused, and the supplied rules tuple should remain the exact shared object. Check one access path at a time.

Hint 2: match each function to one ownership boundary

`record_rune_in_place` follows the supplied explorer and mutates its three relevant descendants. `with_supply` first calls the same safe spawning/copying logic with the explorer's existing name, then appends only to the new inventory. The snapshot converts internal lists to tuples. The history helper constructs a list only when its parameter is `None`.

Hint 3: assemble the selective copy and artifact

Pseudocode:

```text
new explorer = shallow outer copy
replace name
copy inventory
create journal dictionary with copied runes and route
copy visited set
reuse rules tuple deliberately

message = uppercase name + vault phrase + length of runes + plural noun
```

The shallow variation leaks because its new outer dictionary still holds the source journal reference. `spawn_explorer` repairs every mutable path.

7. Keep debugging evidence

Preserve one failed assertion from before your repair. Connect it to one exact path rather than writing only "the copy was wrong."

Failure	Value and identity evidence	Ownership hypothesis	Controlled change	Verified rerun
Which numbered assertion failed?	What were `==`, `is`, and relevant contents?	Which outer or nested object is shared incorrectly?	Which one construction/copy/mutation changed?	Which earlier and new assertions now pass?

Useful failures include a shared journal dictionary, independently copied journal with a still-shared rune list, a returned internal list, a reused default history, or a list room marker rejected as unhashable.

8. Compare with a complete solution

Open this only after attempting each assertion group and using hints in order.

Show one complete Mirror Vault solution

```python
def spawn_explorer(blueprint, *, name):
    """Return an explorer with independent mutable state and shared rules."""
    explorer = blueprint.copy()
```

PYTHON (CONTINUED)

```
    explorer["name"] = name
    explorer["inventory"] = blueprint["inventory"].copy()
    explorer["journal"] = {
        "runes": blueprint["journal"]["runes"].copy(),
        "route": blueprint["journal"]["route"].copy(),
    }
    explorer["visited"] = blueprint["visited"].copy()
    explorer["rules"] = blueprint["rules"]
    return explorer


def identity_report(left, right):
    """Return labeled identity evidence for important explorer paths."""
    return {
        "same_outer": left is right,
        "same_inventory": left["inventory"] is right["inventory"],
        "same_journal": left["journal"] is right["journal"],
        "same_runes": left["journal"]["runes"] is right["journal"]["runes"],
        "same_route": left["journal"]["route"] is right["journal"]["route"],
        "same_visited": left["visited"] is right["visited"],
        "same_rules": left["rules"] is right["rules"],
    }


def record_rune_in_place(explorer, room, rune):
    """Record room and rune in caller-owned explorer; return None."""
    explorer["visited"].add(room)
    explorer["journal"]["route"].append(room)
    explorer["journal"]["runes"].append(rune)


def with_supply(explorer, supply):
    """Return independent explorer state containing one additional supply."""
    updated = spawn_explorer(explorer, name=explorer["name"])
    updated["inventory"].append(supply)
    return updated


def journal_snapshot(explorer):
    """Return an immutable snapshot of rune and route sequences."""
    return {
        "runes": tuple(explorer["journal"]["runes"]),
        "route": tuple(explorer["journal"]["route"]),
    }


def remember_attempt(attempt, history=None):
    """Append attempt to supplied history or a fresh per-call list; return it."""
    if history is None:
        history = []
    history.append(attempt)
    return history


def vault_message(explorer):
    """Return the explorer's derived Mirror Vault escape message."""
    name = explorer["name"].upper()
    rune_count = len(explorer["journal"]["runes"])
    return f"{name} BREAKS THE MIRROR VAULT WITH {rune_count} RUNES"
```

9. Predict another ownership change

Suppose the vault adds mutable `settings = {"sound": True}` to the blueprint. Before editing code, predict:

- whether the current `spawn_explorer` shares that dictionary;

- which identity-report field you would add;
- whether settings should be shared or independently editable;
- the minimum copy change for that ownership decision; and
- which source-preservation assertion should fail before the repair.

This variation checks whether you can extend the object graph rather than merely remember the existing field names.

10. Check your understanding

Answer from the completed object graph and assertions rather than the story alone.

Mirror Vault evidence check

Trace ownership, mutation, stable keys, and returned state.

1. Why does leaky_clone change leaky_source’s rune list?

- **A.** Equal strings synchronize
- **B.** Sets mutate dictionaries
- **C.** The outer copy deep-copied the journal
- **D.** Both outer dictionaries still reach the same journal and rune list

2. Why is `(1, 0)` suitable for the visited set?

- **A.** The tuple contains hashable integers and has stable lookup behavior
- **B.** Its printed hash is permanent
- **C.** All coordinates are immutable globals
- **D.** Set membership uses identity only

3. What makes with_supply source-preserving?

- **A.** It removes the source name
- **B.** It creates independent mutable explorer state before appending
- **C.** Append returns a deep copy
- **D.** The rules tuple copies the inventory

4. Why return tuples from journal_snapshot?

- **A.** Tuples turn rooms into hashes automatically
- **B.** Dictionaries cannot return lists
- **C.** Callers cannot append through them into the explorer’s internal lists
- **D.** Tuple values rebind the explorer

Write your choices before checking the answer key at the end of this book.

11. Decide whether the challenge is complete

Evidence rubric

Evidence	Ready to record when
Object graph	You can identify every intentionally shared and independently owned path.
Behavior	All 24 assertions pass unchanged from a clean state.
Mutation API	The in-place command changes only its supplied explorer and returns `None`.
Copy API	The transformer returns independently editable state and preserves its source.
Boundaries	Tuple markers, safe snapshots, and fresh default history behave as documented.
Debugging	One failed assertion is connected to a precise shared path and verified repair.
Artifact	The final sentence is derived from Nova’s name and collected rune count.

Record completion only when every statement is true:

- ☐ All 24 supplied assertions pass without editing expected values.
- ☐ I can draw the blueprint, Nova, and Sol mutable descendants separately.
- ☐ The immutable rules relationship is intentionally shared.
- ☐ The final artifact is derived rather than assigned directly.
- ☐ I reproduced and repaired the shallow-copy curse.
- ☐ I predicted one schema/ownership variation before implementing it.
- ☐ I kept one complete debugging record.
- ☐ The artifact works after restarting and running every cell in order.

This button stores a self-reported marker only in this browser. It does not submit the artifact, grade it, verify identity, or issue a certificate.

Key points

- Object graphs make accidental sharing observable before a mutation spreads.
- Selective copying can isolate every mutable descendant while safely reusing an immutable value whose sharing is intentional.
- Hashable tuple coordinates make visited-room membership stable and meaningful.
- In-place commands, returned-copy transformers, snapshots, and optional defaults each need explicit ownership contracts.

- A shallow-copy failure is useful evidence when it identifies the exact nested reference that remained shared.
- The successful escape sentence is derived from independently owned state, not hidden directly in the implementation.

Milestone Challenge: Route Supplies through a Puzzle Village

Online lesson and runnable notebook

Combine collections, decisions, loops, functions, and copying to allocate limited supplies.

- **Goal:** Combine collections, decisions, loops, functions, and copying to allocate limited supplies.
- **Estimated time:** 2–4 hours of active work.
- **Before starting:** Complete the milestone units and their unit challenges.
- **Keep:** Your source, observed outputs, checks, and a debugging note.

This community challenge is ungraded. Use the checks to review your own work; there is no submission service or automatic grade. Completion is self-reported in this browser on this device.

1. Route supplies through a puzzle village

Implement allocate(stock, requests). stock maps names to nonnegative integer quantities. Each request is an (item, quantity) pair with a positive integer quantity. Process requests in order. Accept only when the full quantity is available; otherwise reject without changing stock. Unknown items are rejected. Return (remaining, decisions): a new stock dictionary and a list of booleans. Do not mutate either input. Treat valid quantities as preconditions.

Predict an ordinary result before coding. Start with the smallest part you can run and inspect. Use Colab for notebooks and the local workspace introduced in the units for packaging or command-line tasks.

2. Set up the experiment

PYTHON

```
def allocate(stock, requests):
    """Return independent remaining stock and one decision per request."""
    # Implement the allocation contract.
    pass
```

The starter is deliberately incomplete. Implement the contract; do not change the expected results to match an incorrect implementation.

3. Check the observable results

1. For {"rope": 3, "lamp": 1} and [("rope", 2), ("rope", 2), ("lamp", 1)], return ({"rope": 1, "lamp": 0}, [True, False, True]).
2. Empty requests return an equal but independent stock dictionary and an empty decisions list.
3. An unknown item is rejected without adding a stock entry.
4. After the call, the original stock is unchanged. Mutating remaining must not alter it. Explain why a shallow copy is sufficient for integer values.

Record actual outputs beside predictions. Reduce a failure to a small example, change one cause at a time, and repeat the failed check after repairing it.

4. Change one condition

Change one rope request to quantity 1, reverse the two rope requests, and trace the decisions. Explain why order is part of the contract.

5. Review a hint after attempting the task

Hint: connect the relevant tools

Begin with stock.copy(). Compare each request with remaining.get(item, 0), append its decision, and subtract only on acceptance. Return after processing every request.

6. Explain and self-review

Keep a trace table, assertions for ordinary and boundary cases, and an explanation of one aliasing bug. Show exactly which object changes at each step.

Review path: compare your result with the contract

Check the ordinary result first, then the boundary cases and changed condition. Explain which rule justifies each result. An implementation that differs from the hint can still satisfy the contract; explain why yours does.

Check the reasoning behind your result

Choose an answer, then review the explanation.

1. Why does remaining = stock violate the contract?

- **A.** It sorts the requests.
- **B.** Both names refer to the same mutable dictionary.
- **C.** Integers cannot be dictionary values.
- **D.** It makes a deep copy.

Write your choices before checking the answer key at the end of this book.

7. Save your work and record completion

Save the notebook or project somewhere you control. Include one decision, one failed check, its repair, and a result you can reproduce. Record completion after reviewing the checks; you can undo it later.

Return to the milestone overview.

Build Reliable Programs from External Data

Online lesson and runnable notebook

Build a reproducible package that parses, validates, and summarizes a partly damaged CSV log.

What you will build

Build a reproducible package that parses, validates, and summarizes a partly damaged CSV log.

Follow the units in order and complete each unit challenge. Keep examples and debugging notes that help explain your choices. This overview is orientation, not a scored activity.

Unit sequence

1. Solving Problems with Algorithms — 4 lessons and a unit challenge.
2. Errors, Exceptions, and Debugging — 4 lessons and a unit challenge.
3. Files, Paths, and External Data — 5 lessons and a unit challenge.
4. Modules, Environments, and Python Projects — 6 lessons and a unit challenge.

Prepare and pace your work

Complete the preceding milestone in the course sequence, or review its outcomes if you already have that experience. Use each unit's active-time estimate and pause at lesson boundaries. Plan another 2–4 hours for the milestone challenge, including rerunning checks and explaining your result.

Use Colab for notebook exercises and the workspace introduced in the units for local environment, packaging, and command-line tasks.

Connect the units

Finish with Recover the Observatory Log. Combine the milestone skills and check a changed condition. The challenge provides a contract, expected results, a hint, and self-review guidance.

Community learning is ungraded. Completion controls record your own progress only in this browser on this device; they do not issue a verified credential.

Begin the first unit.

Unit 7: Solving Problems with Algorithms

Online lesson and runnable notebook

Learn a repeatable route from an unclear request to a small, checkable algorithm whose result and practical cost you can explain.

- **Level:** Python Foundations · Unit 7
- **Estimated time:** 16–22 hours including the challenge
- **Unit outcome:** Turn an unclear task into explicit examples and rules; decompose it into checkable functions; select, trace, and adapt common algorithms; and defend correctness, boundary behavior, and practical time and memory costs.
- **Practice in:** Generated Colab notebooks or a local Python project

1. One request can hide several different programs

Imagine that a friend asks you to write a program that will **choose the best stargazing site**. Python cannot act on that sentence yet. Does *best* mean the site with the darkest sky, the shortest drive, the highest visitor rating, or some combination? If two sites are equally good, which one wins? What happens when every site is closed?

All of these answers are reasonable:

```python
sites = [
    {"name": "Pine Ridge", "darkness": 9, "minutes": 55},
    {"name": "Moon Lake", "darkness": 8, "minutes": 20},
]

best_darkness = "Pine Ridge"
shortest_drive = "Moon Lake"
```

The difficulty is not Python syntax. The request does not yet contain enough information to decide which result is correct. Unit 7 teaches you to notice that problem before assumptions become code.

A correct program needs a checkable promise

An algorithm is not correct merely because it runs or because its output looks reasonable. It is correct when its observable behavior agrees with an explicit contract for every supported input. :::

2. A small example is the first design tool

Suppose the clarified rule is:

Choose the open site with the greatest darkness score. If scores tie, choose the shorter drive. If no site is open, return `None`.

Now a small example can reveal the decision:

```python
sites = [
    {"name": "Pine Ridge", "darkness": 9, "minutes": 55, "open": True},
    {"name": "Moon Lake", "darkness": 9, "minutes": 20, "open": True},
]

expected_name = "Moon Lake"
```

Both sites have the same darkness score, so the travel-time rule decides the result. A one-site example could not reveal that rule. A useful example is not necessarily large; it is just large enough to distinguish correct behavior from a tempting wrong behavior.

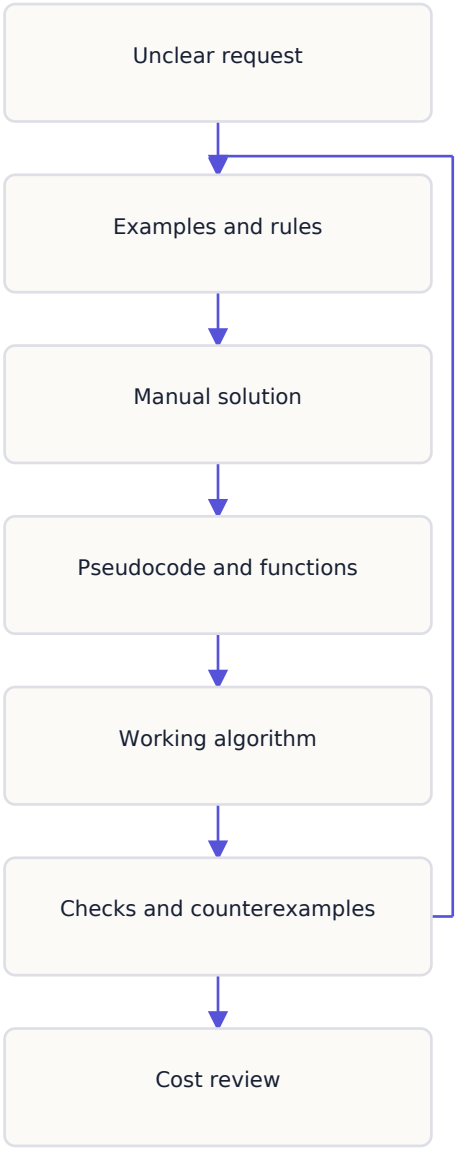
During this unit, you will repeatedly use a table like this:

Case	Input feature	Expected result	Rule exposed
Ordinary	Different darkness scores	Darker site	Main priority
Tie	Same darkness, different travel	Shorter drive	Tie rule
Boundary	One open site	That site	Smallest result-bearing input
No result	Every site closed	`None`	Empty candidate behavior

3. Follow the problem-solving loop

You do not need a brilliant idea before you begin. You need a series of small, observable decisions.

The diagram follows an unclear request through specification, implementation, and review; a failed check sends you back to the earliest unsupported decision.



Checks and counterexamples → Examples and rules: Rule or plan is wrong

The order matters:

1. **Examples and rules** define what an answer means.
2. **A manual solution** exposes the decisions and changing state.
3. **Pseudocode and functions** give those responsibilities names.
4. **A working algorithm** implements one checkable stage at a time.
5. **Checks and counterexamples** challenge the result rather than admire it.
6. **A cost review** asks how work and memory grow with larger inputs.

You may move backward when new evidence reveals a missing rule. That is not failure. It is how a vague request becomes dependable software.

4. Four lessons answer four different questions

Step	Lesson	Question answered	Work you will retain
1	Turn a Vague Idea into Testable Examples	What exactly must the program do?	Contract, acceptance table, and counterexamples
2	Turn a Plan into Working Python	How can a hand solution become small functions?	Trace, pseudocode, interfaces, and staged checks
3	Choose an Algorithm That Fits	Which traversal, state, and stopping rule match the result?	Pattern decision table and reusable implementations
4	Know It Works and Understand the Cost	Why is the result correct, and how does the work grow?	Invariant, operation counts, and trade-off explanation
5	Unit Challenge	Can you combine the complete method independently?	A working observatory puzzle and debugging record

Each lesson has its own main task. Short checkpoints appear near the ideas they assess, and the final lab asks you to combine the page rather than repeat its opening code.

5. Bring the tools from Units 3–6

You already know most of the Python vocabulary this unit needs:

- **Unit 3** supplied lists, dictionaries, sets, sorting, and nested records.
- **Unit 4** supplied decisions, loop state, stopping, searches, and comprehensions.
- **Unit 5** supplied function contracts, parameters, return values, and decomposition.
- **Unit 6** supplied ownership decisions, non-mutation promises, and ways to recognize shared state.

This unit changes the question. Instead of “How does a `for` loop execute?” you will ask “Does this result require every item, or may the loop stop?” Instead of “How do I define a function?” you will ask “Which responsibility can be checked independently?”

Keep later tools in their later units

Unit 8 develops exception handling, traceback reading, debugger use, and a full evidence-driven debugging process. Unit 13 develops pytest and test-suite design. Here you will use plain assertions, small counterexamples, and value traces so that those later tools have clear behavior to protect. :::

6. Prepare a problem-solving notebook

Create these headings in a new notebook or text file:

```text
1. Request
2. Questions and chosen rules
3. Acceptance examples
4. Manual trace
5. Pseudocode
```

```text (continued)
6. Function contracts
7. Implementation
8. Checks and counterexamples
9. Time and extra-space notes
10. Debugging record
```

Do not try to fill every section immediately. Move down the list as decisions become stable, and move back up when a check exposes ambiguity.

Use assertions for exact observations:

```python
actual = 7
expected = 7
assert actual == expected
```

An assertion is useful only when `expected` came from the contract or a manual calculation. Writing `expected = actual` would make the check pass while proving nothing.

7. Choose a realistic pace

This unit contains about **16–22 hours of active work**. That estimate includes typing, predicting, tracing, quizzes, labs, and the challenge—not only reading. A useful pace is:

- **Session 1:** clarify requirements and build acceptance examples;
- **Session 2:** write pseudocode and implement a small pipeline;
- **Sessions 3–4:** learn, compare, and combine algorithm patterns;
- **Session 5:** reason about correctness and practical cost;
- **Session 6:** complete the unit challenge from a clean state.

Stop at a point where you can leave a short note describing the next check. On returning, rerun the notebook from the top before trusting old output.

8. Preview the clockwork observatory

The unit challenge gives you a signal and matching energy readings:

```python
signal = "AABCDEFAC"
energy_readings = [2, 1, 3, 2, 4, 1, 5, 2, 1]
```

You will derive three pieces of a wake-up code:

- the end position of the first fixed-width window containing distinct symbols;
- the first step where accumulated energy reaches a target; and
- the most frequent anchor symbol, with a stated tie rule.

The challenge supplies function names, docstrings, progressive assertions, three hint levels, and a hidden solution. It does not choose the loop state or conditions for you. Completing the four lessons first should make each decision recognizable rather than mysterious.

9. Start by making “best” precise

You are ready when you can run ordinary list, loop, dictionary, set, and function examples from earlier units. You do not need to remember every method; you do need to inspect a small case carefully and explain what your code reads, changes, and returns.

Continue to Turn a Vague Idea into Testable Examples.

Key points

- Ambiguous requests must become observable rules before code can be judged.
- Small examples are design tools when each one exposes a distinct rule.
- Manual work, pseudocode, functions, checks, and cost reasoning form one connected process.
- This unit selects and defends algorithms; it does not repeat earlier syntax lessons or anticipate later exception and testing tools.

Turn a Vague Idea into Testable Examples

Online lesson and runnable notebook

Clarify an ambiguous request by defining inputs, outputs, rules, ties, boundaries, constraints, and independently calculated acceptance examples.

- **Level:** Python Foundations
- **Estimated time:** 3–4 hours
- **You will learn:** Turn a vague request into a compact, observable contract and a set of non-redundant acceptance examples.
- **Practice in:** Google Colab, JupyterLab, or a local editor

A program can be beautifully formatted, run without errors, and still answer the wrong question. Before choosing a loop or function, you need to know what result would count as correct.

In this lesson, you will clarify a program that chooses a stargazing site. Keep these questions visible while you work:

1. What information enters the program, and what exactly comes back?
2. Which words in the request hide an unstated decision?
3. Which small input would distinguish two competing interpretations?
4. What happens at a tie, threshold, empty input, or no-result case?
5. Can the expected result be calculated without trusting unfinished code?

1. “Choose the best site” has more than one answer

Start with two site records:

```python
sites = [
    {
        "name": "Pine Ridge",
        "darkness": 9,
        "minutes": 55,
        "accessible": True,
        "open": True,
    },
    {
        "name": "Moon Lake",
        "darkness": 8,
        "minutes": 20,
        "accessible": True,
        "open": True,
    },
]
```

Read each key as a fact the program may use. `Pine Ridge` has a darker sky, but `Moon Lake` is closer. Either name could be a sensible answer to “best.” Running more Python will not settle a decision the request never made.

Write two predictions before continuing:

```python
best_by_darkness = "Pine Ridge"
best_by_travel_time = "Moon Lake"
```

Both predictions agree with the input. They disagree because they implement different rules.

A reasonable assumption is still an assumption

When a rule affects observable output, record it or ask for clarification. Do not bury it inside a comparison and hope every reader would choose the same meaning. :::

2. Mark the data, action, and missing rules

Consider this fuller request:

From the open, accessible sites, choose the site with the greatest darkness score. If darkness ties, choose the shorter drive. Return its name. If no site qualifies, return `None`. Do not change the supplied records.

The nouns suggest data:

- **sites**: a collection of candidate records;
- **open** and **accessible**: eligibility facts;
- **darkness** and **minutes**: comparison values;
- **name**: the returned text.

The verbs and comparison phrases suggest behavior:

- **choose** one site;
- **from** only qualifying candidates;
- **greatest** darkness first;
- **shorter** drive on a darkness tie;
- **return** a name or `None`;
- **do not change** the caller’s data.

This annotation does not determine the exact loop. It determines what any loop must accomplish.

Separate product rules from implementation choices

These are product rules because changing them changes observable behavior:

```python
empty_result = None
darkness_priority = "greatest"
tie_priority = "shortest drive"
```

These are implementation choices because several versions could preserve the same behavior:

```python
possible_strategy_a = "keep a running best candidate"
possible_strategy_b = "filter and sort a new list"
```

Lesson 3 compares strategies. For now, do not put an implementation detail such as “must sort” into the contract unless there is a real external reason.

3. Ask questions before assumptions become code

A short requirements interview can prevent a long repair. For the site chooser, ask:

Question	Why the answer matters
Is a larger darkness score always better?	Establishes comparison direction
Must a site be both open and accessible?	Establishes eligibility logic
What decides an exact tie on darkness and travel?	Prevents unstable output
Is an empty list supported?	Establishes no-result behavior
May the function reorder or edit the list?	Establishes ownership behavior
Do all records contain the named keys?	Establishes a precondition
About how many sites are expected?	May influence later cost choices

Assume the answer to the exact-tie question is: **keep the site that appears first in the input**. That rule matters even if the sample data has no exact tie.

Ask the question that could change an assertion

A useful clarification changes an input, an expected result, or a supported behavior. Editor theme, internal variable names, and preferred line count do not belong in this contract. :::

4. State the contract in observable language

A compact contract can live in prose before it becomes a docstring:

- **Input:** a list of site dictionaries containing `name`, `darkness`, `minutes`, `accessible`, and `open`.
- **Precondition:** the fields already contain supported values; detailed validation is outside this function.
- **Eligibility:** both `open` and `accessible` must be true.
- **Selection:** greatest darkness, then shortest drive, then earliest input position.
- **Return:** the selected site’s name, or `None` when no site qualifies.
- **Ownership:** reading is allowed; changing the list or its records is not.

A future function interface can express the same promise:

```python
def choose_site(sites):
    """Return the best eligible site name, or None when no site qualifies."""
    pass
```

`pass` is a placeholder, not an implementation. The valuable work at this stage is the promise surrounding it.

5. Calculate one ordinary result by hand

Use candidates with different darkness scores so the main rule decides:

```python
ordinary_sites = [
    {
        "name": "Moon Lake",
        "darkness": 7,
        "minutes": 20,
        "accessible": True,
        "open": True,
    },
    {
        "name": "Pine Ridge",
        "darkness": 9,
        "minutes": 55,
        "accessible": True,
        "open": True,
    },
]

ordinary_expected = "Pine Ridge"
```

The manual calculation is:

1. both records satisfy the two eligibility conditions;
2. darkness scores are `7` and `9`;
3. `9` is greater, so the travel-time tie rule is not used;
4. the promised return shape is a name, so the result is `"Pine Ridge"`.

Later, the acceptance check will be:

```python
assert choose_site(ordinary_sites) == ordinary_expected
```

The assertion may currently fail because `choose_site` contains only `pass`. That is expected. The check describes a target before the implementation exists.

6. Use the smallest case that distinguishes two rules

A one-site input can check return shape, but it cannot distinguish the main rule from the tie rule. Use two equally dark sites:

```python
tie_sites = [
    {
        "name": "Pine Ridge",
        "darkness": 9,
        "minutes": 55,
        "accessible": True,
        "open": True,
    },
    {
        "name": "Moon Lake",
        "darkness": 9,
        "minutes": 20,
        "accessible": True,
        "open": True,
```

PYTHON (CONTINUED)

```
    },
]

tie_expected = "Moon Lake"
```

If an implementation chooses only the greatest darkness value and keeps the first match, it returns `"Pine Ridge"`. This example exposes the missing travel rule with only two records.

Now isolate the final tie rule:

PYTHON

```
exact_tie_sites = [
    {
        "name": "North Field",
        "darkness": 8,
        "minutes": 30,
        "accessible": True,
        "open": True,
    },
    {
        "name": "South Field",
        "darkness": 8,
        "minutes": 30,
        "accessible": True,
        "open": True,
    },
]

exact_tie_expected = "North Field"
```

The cases are small because each one answers one design question.

Checkpoint: rules that produce observable results

Rules and observable behavior

Choose the answer that follows from the stated site contract.

1. Two eligible sites both have darkness 9. One is 20 minutes away and one is 50 minutes away. Which rule decides?

- **A.** Input order immediately
- **B.** The shorter-drive tie rule
- **C.** Alphabetical name order
- **D.** Whichever dictionary has more keys

2. Which statement is an implementation choice rather than an observable product rule?

- **A.** Closed sites are excluded.
- **B.** No qualifying site returns None.
- **C.** Use a running-best loop rather than sorting.
- **D.** An exact tie keeps the earlier record.

3. What does the site chooser return for a successful selection?

- **A.** The entire input list

- **B.** The selected site’s name
- **C.** Every darkness score
- **D.** A printed trace with no returned value

Write your choices before checking the answer key at the end of this book.

7. Boundaries appear where behavior can change

A boundary is not merely an unusual value. It is a point near which the rule or result may change. For eligibility, the boundary is the change from `False` to `True`. For a numeric threshold such as “darkness at least 7,” check values just below, exactly at, and just above the threshold.

```python
threshold_scores = [6, 7, 8]
expected_eligible = [False, True, True]
```

For the current site contract, useful boundaries include:

```python
empty_sites = []
empty_expected = None

one_site = [
    {
        "name": "Solo Hill",
        "darkness": 5,
        "minutes": 10,
        "accessible": True,
        "open": True,
    }
]
one_expected = "Solo Hill"
```

A no-result input need not be empty. It can contain records that all fail the eligibility rule:

```python
unavailable_sites = [
    {
        "name": "Closed Bluff",
        "darkness": 10,
        "minutes": 5,
        "accessible": True,
        "open": False,
    },
    {
        "name": "Rough Trail",
        "darkness": 10,
        "minutes": 5,
        "accessible": False,
        "open": True,
    },
]

unavailable_expected = None
```

These two no-result cases reach the same output through different input conditions. Both are useful because they can expose different mistakes.

8. Counterexamples challenge an incomplete rule

Suppose someone proposes:

Return the site with the greatest darkness score.

This input is a counterexample:

```python
counterexample_sites = [
    {
        "name": "Locked Peak",
        "darkness": 10,
        "minutes": 15,
        "accessible": True,
        "open": False,
    },
    {
        "name": "Open Meadow",
        "darkness": 7,
        "minutes": 25,
        "accessible": True,
        "open": True,
    },
]

counterexample_expected = "Open Meadow"
```

The greatest darkness belongs to an ineligible site. The input is more useful than saying “remember eligibility” because it can be run against an implementation.

A counterexample should be as small as practical. Extra records can hide the one relationship that disproves the rule.

9. Expected values need an independent source

This check looks official but proves nothing:

```python
actual_total = sum([4, 7, 9])
expected_total = sum([4, 7, 9])
assert actual_total == expected_total
```

If the calculation were wrong for a subtle reason, both sides would repeat the same mistake. Instead, calculate a small expected result separately:

```python
values = [4, 7, 9]
expected_total = 20
actual_total = sum(values)
assert actual_total == expected_total
```

For a more complex rule, keep a short note beside the expected value:

```python
# Both are equally dark; 18 minutes beats 42 minutes.
expected_name = "River Bend"
```

Independent does not mean another large program. It often means arithmetic by hand, a tiny table, an agreed example from a stakeholder, or a simpler reference method used only on small inputs.

10. Constraints must be concrete enough to affect a design

“Make it fast” gives no usable target. Compare these statements:

- The program usually receives **fewer than 20 sites**.
- The program may receive **one million sites**.
- The result must be returned **without changing the input order**.
- The program may use **one additional collection proportional to the input**.
- The function is called **once per evening**, not thousands of times per second.

These facts can affect later choices. With 20 sites, a direct, easy-to-explain solution may be preferable even if another strategy has better growth. With one million records, repeated scans deserve closer attention.

Do not confuse a constraint with a premature command:

```text
Useful: preserve the caller's order and support up to 100,000 records.
Premature: use a set and exactly one loop.
```

The useful statement describes observable behavior and scale. The premature statement chooses tools before alternatives have been compared.

Checkpoint: boundaries and trustworthy expectations

Boundaries and acceptance evidence

Identify the case that reveals the named behavior.

1. For the rule "darkness must be at least 7," which set best checks the numeric boundary?

- **A.** 1, 2, and 3
- **B.** 7, 70, and 700
- **C.** 6, 7, and 8
- **D.** Only 10

2. Why keep both an empty case and a non-empty case where every site is ineligible?

- **A.** They exercise different paths that should reach the same promised result.
- **B.** Python requires two None values.
- **C.** The larger case always proves better performance.
- **D.** They force the function to sort.

3. Which expected value is the strongest evidence for a small total?

- **A.** expected = actual
- **B.** expected = the same unfinished function called again

- **C.** expected = a random nearby number
- **D.** expected = 20, calculated by hand for [4, 7, 9]

Write your choices before checking the answer key at the end of this book.

11. Build an acceptance table before implementation

Combine the examples into one compact specification:

Case	Important input feature	Expected	Rule exercised
ordinary	eligible sites with different darkness	`"Pine Ridge"`	greatest darkness
darkness tie	equal darkness, different travel	`"Moon Lake"`	shortest drive
exact tie	equal darkness and travel	`"North Field"`	earliest input
one site	one eligible record	`"Solo Hill"`	smallest successful input
empty	no records	`None`	no candidate
all unavailable	records exist but none qualify	`None`	eligibility
closed but darker	ineligible record would otherwise win	`"Open Meadow"`	filter before selection

Ask whether each row could catch a bug that the others might miss. If not, combine or remove redundant cases. More cases are not automatically better; each should have a reason.

Represent the stable table as data when you are ready to implement:

```python
acceptance_cases = [
    (ordinary_sites, "Pine Ridge"),
    (tie_sites, "Moon Lake"),
    (exact_tie_sites, "North Field"),
    (one_site, "Solo Hill"),
    (empty_sites, None),
    (unavailable_sites, None),
    (counterexample_sites, "Open Meadow"),
]
```

Lesson 2 will use this collection to guide incremental implementation.

12. Repair specifications that cannot be checked

Rewrite each weak statement before opening the suggested repair.

“Return a useful answer”

Show one checkable repair

Return the selected site’s `name` string, or `None` if no site is both open and accessible.

“Choose the first good site”

Show the missing questions

Define what *good* means, whether the input order is meaningful, and whether a later site can replace the first one. If the intended contract is first-match search, state the exact condition and no-match result.

“The program should be efficient”

Show a measurable replacement

State the expected input range and important resource constraint. For example: “Support up to 100,000 site records without modifying them; compare one-pass and sorted-copy strategies before choosing.”

13. Lab: specify a moon-base supply selector

A moon base receives supply-pod records:

```python
pods = [
    {"id": "LUNA-4", "oxygen": 8, "mass": 6, "damaged": False},
    {"id": "NOVA-2", "oxygen": 8, "mass": 4, "damaged": False},
    {"id": "DUST-9", "oxygen": 10, "mass": 3, "damaged": True},
]
```

The request is deliberately vague:

Choose the best usable supply pod and show its ID.

Do not implement the selector yet. Produce:

1. a list of clarification questions;
2. a chosen rule for usability;
3. a primary comparison and at least two tie rules;
4. an exact return shape and no-result behavior;
5. a non-mutation promise;
6. an expected input-size statement; and
7. at least six acceptance examples covering ordinary, boundary, tie, and counterexample behavior.

One possible rule set

If you need a starting direction, choose undamaged pods, prefer greater oxygen, then lower mass, then earlier input position. Return the ID or `None`. :::

Hint: choose cases that can disagree

Include a damaged pod with the highest oxygen, two undamaged pods with equal oxygen and different mass, an exact tie, one usable pod, no records, and only damaged records.

Show a model acceptance strategy

A strong table states that a pod is usable when `damaged` is `False`; greater `oxygen` wins; lower `mass` breaks an oxygen tie; earlier input breaks an exact tie; the return value is an ID or `None`; and input records

remain unchanged. Calculate every expected ID by hand and write one sentence naming the rule each case isolates.

14. Check the complete specification

Run this checklist against both the site chooser and your supply selector:

- ☐ The input and returned result have exact shapes.
- ☐ Every comparison direction is stated.
- ☐ Eligibility, ties, empty input, and no-result behavior are explicit.
- ☐ Ownership and realistic scale are explicit.
- ☐ Each acceptance example exposes a distinct rule.
- ☐ Expected values were calculated independently.
- ☐ Internal loop or sorting choices were not mistaken for product rules.

Checkpoint: a specification ready for implementation

Is the specification ready?

Choose the statement that makes the future implementation checkable.

1. Which pod case best exposes an implementation that forgets to exclude damaged pods?

- **A.** A damaged pod has the greatest oxygen, while an undamaged pod has less.
- **B.** Every pod has a different ID.
- **C.** The list contains three undamaged pods in alphabetical order.
- **D.** Every mass value is even.

2. Why must an exact tie rule be written even if the opening sample contains no exact tie?

- **A.** It makes dictionaries immutable.
- **B.** It determines observable output for another supported input.
- **C.** It guarantees constant-time execution.
- **D.** It removes the need for a no-result case.

3. Which statement gives the clearest design constraint?

- **A.** Make it elegant.
- **B.** Use clever Python.
- **C.** Never write more than five lines.
- **D.** Support up to 100,000 records without changing their order.

Write your choices before checking the answer key at the end of this book.

Key points

- Clarify inputs, outputs, eligibility, priority, ties, no-result behavior, ownership, and scale before choosing implementation details.

- An acceptance example contains explicit input and an independently calculated observable result.
- Ordinary cases show intended behavior; boundaries and counterexamples expose incomplete rules.
- A small case is powerful when it distinguishes two plausible implementations.
- Constraints should describe supported scale or behavior, not prescribe a favorite tool without evidence.

References

- Python tutorial: defining functions and documenting their behavior
- Python reference: the `assert` statement
- Python tutorial: comparing sequences and other values

Turn a Plan into Working Python

Online lesson and runnable notebook

Move from a hand-solved case to traceable pseudocode, cohesive function contracts, and an incremental Python implementation with checks at every stage.

- **Level:** Python Foundations
- **Estimated time:** 3–4 hours
- **You will learn:** Turn a manually solved case into pseudocode, cohesive functions, and a working program built one checkable behavior at a time.
- **Practice in:** Google Colab, JupyterLab, or a local editor

Lesson 1 ended with a contract for choosing a moon-base supply pod. A contract says what must be observable; it does not yet say how the program will reach the result. This lesson supplies that bridge.

Keep these questions near your code:

1. What did you do by hand to solve one small case?
2. Which value must survive while you inspect later records?
3. Where can the work be split into independently checkable responsibilities?
4. Does the pseudocode show every decision, update, and stopping point?
5. What is the smallest end-to-end behavior you can make pass now?

1. Solve one supply case without Python

Use the rule set from Lesson 1:

- ignore damaged pods;
- prefer greater oxygen;
- on an oxygen tie, prefer lower mass;
- on an exact tie, keep the earlier pod;
- return its ID, or `None` when no pod is usable;
- do not modify the input.

Start with three records:

```python
pods = [
    {"id": "LUNA-4", "oxygen": 8, "mass": 6, "damaged": False},
    {"id": "NOVA-2", "oxygen": 8, "mass": 4, "damaged": False},
    {"id": "DUST-9", "oxygen": 10, "mass": 3, "damaged": True},
]
```

Solve it on paper:

1. `LUNA-4` is usable, so it becomes the current choice.
2. `NOVA-2` is usable and ties on oxygen. Its lower mass replaces `LUNA-4`.

3 `DUST-9` has more oxygen but is damaged, so it cannot replace the choice.

4 Return `"NOVA-2"`.

The manual work reveals two responsibilities—eligibility and comparison—and one piece of persistent state: the current best usable pod.

2. Record what changes after every pod

A trace table makes the state visible:

Current pod	Usable?	Best before	Replace?	Best after
`LUNA-4`	yes	`None`	yes	`LUNA-4`
`NOVA-2`	yes	`LUNA-4`	yes: equal oxygen, lower mass	`NOVA-2`
`DUST-9`	no	`NOVA-2`	no	`NOVA-2`

Notice what the table does **not** require. It does not need a second list of usable pods, and it does not need to sort all records. A running choice is enough for this contract.

Try the exact-tie rule by changing the second record:

PYTHON
```
exact_tie_pods = [
    {"id": "LUNA-4", "oxygen": 8, "mass": 6, "damaged": False},
    {"id": "NOVA-2", "oxygen": 8, "mass": 6, "damaged": False},
]

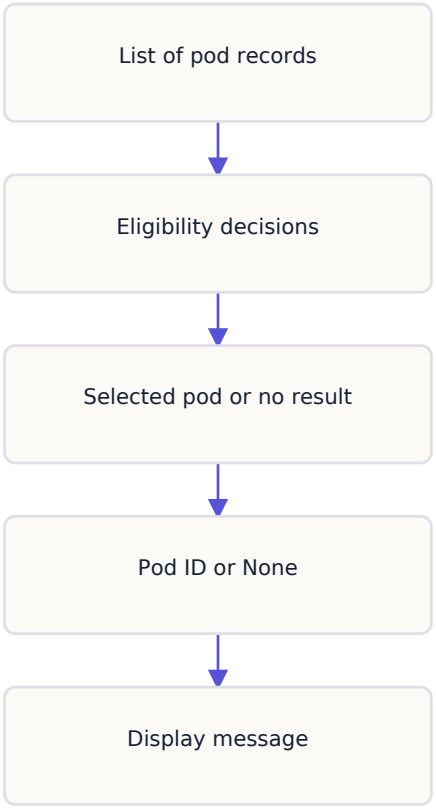
exact_tie_expected = "LUNA-4"
```

The second pod is not better, so the state should remain unchanged. This detail will determine whether the comparison uses `<` or `<=` for mass.

3. Name the shape after every stage

A pipeline is easier to reason about when each arrow has a known data shape.

The supply selector reads pod records, makes a choice, and formats a message without mixing those responsibilities.



You could build a full list of eligible records, but it is not required to find one winner. The conceptual stages still matter even when one loop combines the first two:

```text
records -> eligibility decision -> running best -> returned ID -> message
```

Data-shape notes catch mismatches early. Returning an entire dictionary when the contract promises a string is wrong even if the selected record is correct.

4. Write pseudocode around decisions and state

Pseudocode describes the algorithm without demanding exact Python punctuation:

```text
SET best pod to no result
FOR EACH pod in the input
    IF the pod is damaged
        SKIP it
    IF there is no best pod yet
        SET best pod to this pod
    OTHERWISE IF this pod has more oxygen
        SET best pod to this pod
    OTHERWISE IF oxygen ties and this pod has lower mass
        SET best pod to this pod
IF there is no best pod
    RETURN no result
RETURN the best pod's ID
```

Useful pseudocode contains:

- the order in which items are visited;

- every decision that can change behavior;
- the state that survives between items;
- the update rule;
- the no-result behavior; and
- the returned result.

This pseudocode is incomplete if it says only “find the best pod.” That phrase renames the goal without explaining how progress occurs.

Trace pseudocode before translating it

Use the table from Section 2. At each row, point to the pseudocode condition that explains `Best after`. If no condition explains a row, the plan is incomplete.

A damaged opening pod exposes another useful case:

```python
damaged_first = [
    {"id": "DUST-9", "oxygen": 10, "mass": 3, "damaged": True},
    {"id": "NOVA-2", "oxygen": 8, "mass": 4, "damaged": False},
]

damaged_first_expected = "NOVA-2"
```

The initial `None` must remain until an eligible record appears.

5. Give each responsibility a checkable interface

The pod program needs only a few functions:

```python
def is_usable(pod):
    """Return True when pod is not damaged."""
    return not pod["damaged"]


def is_better(candidate, current):
    """Return True when candidate should replace current."""
    if candidate["oxygen"] != current["oxygen"]:
        return candidate["oxygen"] > current["oxygen"]
    return candidate["mass"] < current["mass"]


def choose_pod(pods):
    """Return the best usable pod ID, or None."""
    pass


def format_choice(pod_id):
    """Return a message describing the selection result."""
    pass
```

Each docstring names observable behavior. `is_better` assumes both records are usable because `choose_pod` owns eligibility. An exact tie returns `False` because neither comparison uses equality as a reason to replace.

Check the helpers before the whole selector:

```python
assert is_usable(pods[0]) is True
assert is_usable(pods[2]) is False
assert is_better(pods[1], pods[0]) is True
assert is_better(pods[0], pods[1]) is False
```

A helper is valuable when its name identifies a real decision and its result can be checked independently.

6. Avoid a giant function and meaningless fragments

This function mixes selection, formatting, and display:

```python
def announce_pod(pods):
    best = None
    for pod in pods:
        if not pod["damaged"]:
            if best is None or pod["oxygen"] > best["oxygen"]:
                best = pod
    if best is None:
        print("No pod available")
    else:
        print(f"Dispatch {best['id']}")
```

It also forgets the mass tie rule. A learner checking the printed message must determine whether the error came from eligibility, comparison, or formatting.

The opposite extreme is not better:

```python
def pod_oxygen(pod):
    return pod["oxygen"]


def greater_than(left, right):
    return left > right


def pod_id(pod):
    return pod["id"]
```

These wrappers do not name domain decisions. They force a reader to jump among functions without making any responsibility clearer.

A useful split is neither “one function” nor “as many functions as possible.” Split where a name expresses a distinct rule, side effect, or data boundary.

Checkpoint: hand traces and function boundaries

From manual work to function boundaries

Choose the design that preserves the stated pod contract.

1. What state must survive while a running-best algorithm examines later pods?

- **A.** A new copy of every string
- **B.** Only the final printed sentence
- **C.** The Python version number

- **D.** The current best usable pod or None

2. Which helper names a cohesive, independently checkable rule?

- **A.** is_usable(pod)
- **B.** do_line_three(value)
- **C.** greater_than(left, right) for every comparison
- **D.** get_dictionary(record)

3. Why should an exact oxygen-and-mass tie not replace the current pod?

- **A.** Every loop must skip its second item.
- **B.** Dictionaries cannot be compared.
- **C.** The contract says the earlier input record wins an exact tie.
- **D.** None can store only one record.

Write your choices before checking the answer key at the end of this book.

7. Separate calculation from display

A calculation function should return a result that another function can use:

```python
def format_choice(pod_id):
    """Return a message describing the selection result."""
    if pod_id is None:
        return "No usable supply pod"
    return f"Dispatch {pod_id}"
```

Now formatting has direct checks:

```python
assert format_choice("NOVA-2") == "Dispatch NOVA-2"
assert format_choice(None) == "No usable supply pod"
```

Printing can remain at the outer edge:

```python
selected_id = "NOVA-2"
message = format_choice(selected_id)
print(message)
```

The displayed output is useful to a person, while the returned string is useful to assertions, a notebook, a command-line interface, or another function.

8. Build the smallest complete path

Begin with the smallest successful input:

```python
single_pod = [
    {"id": "SOLO-1", "oxygen": 5, "mass": 9, "damaged": False}
]
```

Implement only the structure needed to select it and handle no result:

```python
def choose_pod(pods):
    """Return the best usable pod ID, or None."""
    best = None

    for pod in pods:
        if not is_usable(pod):
            continue
        if best is None:
            best = pod

    if best is None:
        return None
    return best["id"]
```

Run the narrow checks:

```python
assert choose_pod(single_pod) == "SOLO-1"
assert choose_pod([]) is None
```

This is a **vertical slice**: input enters the public function and the promised result comes out. The selector is not complete yet, but it has one complete, observable behavior.

9. Add one acceptance rule at a time

Extend the replacement decision without changing the public interface:

```python
def choose_pod(pods):
    """Return the best usable pod ID, or None."""
    best = None

    for pod in pods:
        if not is_usable(pod):
            continue
        if best is None or is_better(pod, best):
            best = pod

    if best is None:
        return None
    return best["id"]
```

Now run checks in an order that narrows failures:

```python
assert choose_pod(single_pod) == "SOLO-1"
assert choose_pod([]) is None
assert choose_pod(damaged_first) == "NOVA-2"
assert choose_pod(pods) == "NOVA-2"
assert choose_pod(exact_tie_pods) == "LUNA-4"
```

If the fourth check fails but the first three pass, the likely problem is in comparison rather than empty handling or basic eligibility. Progressive checks turn a long task into a sequence of smaller questions.

10. Find the earliest stage with a wrong value

Suppose the full result is `"LUNA-4"` instead of `"NOVA-2"`. Do not edit every function. Inspect in dependency order:

```python
print(is_usable(pods[1]))
print(is_better(pods[1], pods[0]))
print(choose_pod(pods))
print(format_choice(choose_pod(pods)))
```

Expected observations are:

```text
True
True
NOVA-2
Dispatch NOVA-2
```

The earliest wrong observation points to the smallest responsibility worth investigating. Unit 8 will add traceback reading and debugger tools; here the important habit is to inspect the data where it first becomes incorrect.

Do not change several stages at once

If you rewrite eligibility, comparison, selection, and formatting together, a passing result will not reveal which change repaired the behavior. Restore the last understood state and change one cause. :::

11. Protect the caller’s records

The running-best implementation only reads `pods`. Confirm that promise:

```python
original_pods = [
    {"id": "A", "oxygen": 4, "mass": 8, "damaged": False},
    {"id": "B", "oxygen": 7, "mass": 6, "damaged": False},
]

before = [pod.copy() for pod in original_pods]
choose_pod(original_pods)
assert original_pods == before
```

A sorted approach can also preserve the input if it creates a new list:

```python
ordered = sorted(
    original_pods,
    key=lambda pod: (-pod["oxygen"], pod["mass"]),
)
```

Calling `original_pods.sort(...)` would reorder the caller’s list. Unit 6 taught why that difference matters; the current contract tells you which operation is allowed.

The running-best version has another advantage: its state directly represents the result being built. Lesson 4 will compare its cost with alternatives.

Checkpoint: incremental implementation

Build and check one behavior at a time

Choose the next action that gives the clearest evidence.

1. What makes the single-usable-pod version a useful vertical slice?

- **A.** It implements every future feature.
- **B.** A real input passes through the public function to the promised output shape.
- **C.** It uses the most lines possible.
- **D.** It avoids returning a value.

2. Eligibility checks pass, but is_better returns False for a candidate that should win. Where should investigation begin?

- **A.** Rewrite the entire notebook.
- **B.** Change the expected final ID.
- **C.** Add unrelated formatting.
- **D.** Inspect is_better and its two inputs.

3. Which operation preserves the caller's list order?

- **A.** ordered = sorted(pods, key=...)
- **B.** pods.sort(key=...)
- **C.** pods.reverse()
- **D.** pods.clear()

Write your choices before checking the answer key at the end of this book.

12. Carry a changed rule through every layer

Change the mass tie rule from "lower mass wins" to "higher oxygen efficiency wins," where efficiency is `oxygen / mass`.

Do not begin by editing `is_better`. Update in this order:

1. contract wording;
2. a two-pod example whose winner changes;
3. hand calculation;
4. pseudocode comparison step;
5. `is_better` implementation;
6. progressive assertions; and
7. explanation of the observed change.

One possible distinguishing case is:

```python
changed_rule_pods = [
    {"id": "WIDE-8", "oxygen": 8, "mass": 8, "damaged": False},
    {"id": "LIGHT-6", "oxygen": 6, "mass": 3, "damaged": False},
]
```

Under the original oxygen-first rule, `WIDE-8` wins. Under pure oxygen-per-mass efficiency, `LIGHT-6` wins because `6 / 3` is greater than `8 / 8`. If the intended new rule still prioritizes oxygen before efficiency, the winner would not change. The ambiguity must be resolved before code.

13. Lab: build a festival schedule planner

A small festival has performance records:

```python
performances = [
    {"name": "Comet Choir", "rating": 8, "minutes": 45, "cancelled": False},
    {"name": "Solar Drums", "rating": 9, "minutes": 60, "cancelled": True},
    {"name": "Aurora Strings", "rating": 8, "minutes": 35, "cancelled": False},
]
```

Use this contract:

- ignore cancelled performances;
- prefer a greater rating;
- on a rating tie, prefer the shorter performance;
- on an exact tie, keep the earlier record;
- return the selected name or `None`;
- do not mutate the input;
- format the result as `"Next stage: <name>"` or `"No performance available"`.

Complete these stages:

1. solve the supplied input by hand;
2. write verb-first pseudocode;
3. draw a trace table with current record and running choice;
4. define `is_available`, `is_better_performance`, `choose_performance`, and `format_performance` contracts;
5. implement one usable record and no-result behavior;
6. add main comparison, tie, and cancellation rules one at a time;
7. run at least six progressive assertions;
8. prove the input remains equal to a preserved copy; and
9. change the tie rule and update every affected layer.

Hint: make comparison independent

Let `choose_performance` own traversal and eligibility. Put only the question “should this usable candidate replace the usable current choice?” inside `is_better_performance`.

Show a complete strategy after attempting the lab

Initialize the running choice to `None`. Skip cancelled records. Replace `None` with the first available record. For later available records, replace when the rating is greater or when ratings tie and duration is shorter. Do not replace an exact tie. Return `None` or the chosen name, then pass that value to a separate formatting function. Check cancellation with a record that would otherwise win.

14. Explain the route from input to message

Use the completed supply selector or festival planner and answer aloud:

- Which function reads the collection?
- Which function owns each decision?
- What value persists between iterations?
- Which line changes that value?
- Why does an exact tie keep the earlier item?
- Which function returns data, and which edge displays it?
- Which check would fail first if eligibility were reversed?
- How do you know the input was not changed?

Checkpoint: a plan ready to adapt

Carry a plan into working Python

Choose the statement that keeps decisions visible and checkable.

1. A tie rule changes. What should be updated before the comparison code?

- **A.** Only the printed message
- **B.** The Python installation
- **C.** The contract and a distinguishing expected example
- **D.** Every unrelated helper

2. Why should choose_performance return a name instead of printing the final sentence itself?

- **A.** Returned data can be checked and reused independently from display.
- **B.** Python functions cannot print.
- **C.** A returned string always sorts the input.
- **D.** Printing guarantees a correct tie rule.

3. Which detail is essential in pseudocode for a running-best algorithm?

- **A.** The editor color scheme
- **B.** The condition that replaces the current best value
- **C.** The final number of Python lines
- **D.** The package release date

Write your choices before checking the answer key at the end of this book.

Key points

- Solve a small case manually to expose decisions, persistent state, and the expected result.
- Pseudocode should show traversal, conditions, updates, stopping, and return behavior without depending on exact Python punctuation.
- Split functions at meaningful rules and side-effect boundaries, not at every line.
- Build a narrow end-to-end behavior, then add one acceptance rule at a time.
- Inspect the earliest incorrect stage and preserve the caller’s ownership contract while implementing.

References

- Python tutorial: defining functions and return values
- Python tutorial: looping techniques for sequences and mappings
- Python documentation: `sorted` returns a new list
- Python tutorial: function annotations and docstrings

Choose an Algorithm That Fits

Online lesson and runnable notebook

Recognize common algorithm patterns from the promised result, then adapt their state, stopping, ordering, tie, and no-result rules to a concrete task.

- **Level:** Python Foundations
- **Estimated time:** 4–5 hours
- **You will learn:** Select, trace, and adapt transform, filter, count, accumulation, search, selection, grouping, uniqueness, neighbor, and window patterns.
- **Practice in:** Google Colab, JupyterLab, or a local editor

You already know how `for`, `if`, lists, dictionaries, and sets behave. The new skill is choosing how those tools should work together when the solution has not been designed for you.

This lesson follows signals sent by lantern towers during a night festival. Keep six questions visible:

1. What shape must the result have?
2. What state must survive between input items?
3. Does every item matter, or may the algorithm stop early?
4. What result represents “nothing found”?
5. Must input order be preserved?
6. Which small boundary would expose a wrong adaptation?

1. Let the promised result suggest a pattern

Different requests need different result shapes:

Request	Result shape	State that usually persists	Stop early?
Normalize every signal	one output per input	output collection	no
Keep only active signals	selected subset	output collection	no
Count red signals	one integer	counter	no
Add energy readings	one combined value	running total	no
Find the first alert	one item or no result	no result / found item	yes
Choose the strongest tower	one best item or no result	running best	usually no
Count each symbol	key-to-count mapping	frequency dictionary	no
Group readings by tower	key-to-values mapping	group dictionary	no
Keep first-seen symbols	ordered collection	result plus seen set	no

Request	Result shape	State that usually persists	Stop early?
Detect changes	relationship per adjacent pair	previous/current pair	no
Find a sync sequence	position or no result	current window	yes

The table is not a code generator. Words such as *best*, *unique*, and *find* still hide decisions. The contract must specify ties, order, eligibility, and a no-result value.

2. Transform when every input produces one output

Lantern symbols arrive in mixed case:

```python
raw_symbols = ["r", "G", "b", "R"]
normalized = []

for symbol in raw_symbols:
    normalized.append(symbol.upper())

print(normalized)
```

The output is:

```text
['R', 'G', 'B', 'R']
```

Trace the relationship:

Input position	Input	Output	Output length
0	`"r"`	`"R"`	1
1	`"G"`	`"G"`	2
2	`"b"`	`"B"`	3
3	`"R"`	`"R"`	4

A transformation normally preserves the number and order of items while changing each value. A comprehension expresses the same contract compactly:

```python
normalized = [symbol.upper() for symbol in raw_symbols]
assert normalized == ["R", "G", "B", "R"]
assert len(normalized) == len(raw_symbols)
```

This version accidentally filters while transforming:

```python
incomplete = [symbol.upper() for symbol in raw_symbols if symbol != "b"]
assert incomplete == ["R", "G", "R"]
```

It is valid only if the contract says some inputs should disappear.

3. Filter when only matching inputs survive

Each reading contains an activation flag:

```python
readings = [
    {"tower": "north", "energy": 4, "active": True},
    {"tower": "west", "energy": 9, "active": False},
    {"tower": "south", "energy": 6, "active": True},
]
```

Keep active records in encounter order:

```python
active_readings = []

for reading in readings:
    if reading["active"]:
        active_readings.append(reading)

assert [reading["tower"] for reading in active_readings] == ["north", "south"]
```

Filtering asks a yes/no question for every item. The selected records do not need to be changed. A comprehension can remain readable when the predicate is short:

```python
active_readings = [reading for reading in readings if reading["active"]]
```

Check empty behavior:

```python
all_inactive = [
    {"tower": "east", "energy": 8, "active": False}
]

assert [reading for reading in all_inactive if reading["active"]] == []
```

An empty selected subset is not the same shape as `None`. The contract promised a list, so an empty list is the natural result.

4. Count matches with one explicit counter

Count red symbols without keeping the red symbols themselves:

```python
symbols = "RGBRRY"
red_count = 0

for symbol in symbols:
    if symbol == "R":
        red_count += 1

assert red_count == 3
```

The counter starts at zero because no input has been processed. Each match adds one. Non-matches leave the state unchanged.

A useful invariant in plain language is:

After each iteration, `red_count` equals the number of `R` symbols in the processed prefix.

Do not confuse **counting items** with **adding their values**:

```python
energy_readings = [4, 9, 6]
reading_count = len(energy_readings)
energy_total = sum(energy_readings)

assert reading_count == 3
assert energy_total == 19
```

5. Accumulate with an initial value that fits the operation

A running total begins at zero:

```python
energy_readings = [4, 9, 6]
running_energy = 0

for energy in energy_readings:
    running_energy += energy

assert running_energy == 19
```

A trace shows why:

Energy	Total before	Total after
4	0	4
9	4	13
6	13	19

Joining text has a different practical pattern. Repeated `+=` can work for small text, but collecting pieces and joining once makes the intended separator explicit:

```python
symbols = ["R", "G", "B"]
message = "-".join(symbols)
assert message == "R-G-B"
```

An initial value must match the contract. Zero is suitable for addition, an empty list for collecting results, and `None` for “no candidate yet.” There is no universal initializer for every accumulation.

6. Search for a first match and stop deliberately

Find the first energy reading at or above an alert threshold:

```python
def first_alert(energies, threshold):
    """Return the first alert value, or None when no value qualifies."""
    for energy in energies:
        if energy >= threshold:
            return energy
    return None
```

Run distinct cases:

```python
assert first_alert([3, 8, 12, 7], 10) == 12
assert first_alert([10, 12], 10) == 10
assert first_alert([2, 4, 6], 10) is None
assert first_alert([], 10) is None
```

Returning immediately is correct because later matches cannot replace the **first** match. The return after the loop handles every no-match path.

Compare three related questions:

```python
energies = [3, 8, 12, 7]

has_alert = any(energy >= 10 for energy in energies)
all_safe = all(energy < 15 for energy in energies)
first_value = first_alert(energies, 10)

assert has_alert is True
assert all_safe is True
assert first_value == 12
```

`any` promises a Boolean, `all` promises a Boolean, and `first_alert` promises a value or `None`. The request determines which shape is useful.

Checkpoint: transform, filter, count, accumulate, and search

Choose a single-traversal pattern

Match the requested result to the state and stopping rule it needs.

1. A task must uppercase every symbol while preserving positions. Which pattern fits?

- **A.** First-match search
- **B.** Frequency counting
- **C.** Transform
- **D.** Running-best selection

2. What state is sufficient for counting red symbols?

- **A.** A sorted copy of every symbol
- **B.** An integer initialized to zero
- **C.** The last exception
- **D.** A separate function for each position

3. Why may first_alert return as soon as one value qualifies?

- **A.** Return always skips the first item.
- **B.** Thresholds make loops constant time.
- **C.** None has already been returned.
- **D.** Later values cannot change which qualifying value was encountered first.

Write your choices before checking the answer key at the end of this book.

7. Select the best item with an explicit tie rule

A strongest-tower request cannot usually stop at the first strong reading. A later tower may be stronger.

PYTHON
```
def strongest_tower(readings):
    """Return the active tower with greatest energy, or None."""
    best = None

    for reading in readings:
        if not reading["active"]:
            continue
        if best is None or reading["energy"] > best["energy"]:
            best = reading

    return best
```

Trace the running best:

PYTHON
```
tower_readings = [
    {"tower": "north", "energy": 4, "active": True},
    {"tower": "west", "energy": 20, "active": False},
    {"tower": "south", "energy": 6, "active": True},
    {"tower": "east", "energy": 6, "active": True},
]

winner = strongest_tower(tower_readings)
assert winner["tower"] == "south"
```

`west` is ineligible. `south` replaces `north`. `east` ties `south` but does not replace it because the condition uses `>` rather than `>=`; the earlier active record wins.

After understanding the contract, a built-in can express part of it:

PYTHON
```
active = [reading for reading in tower_readings if reading["active"]]
winner = max(active, key=lambda reading: reading["energy"], default=None)
assert winner["tower"] == "south"
```

`max` keeps the first maximal item, but the filtering and no-result choice still come from your contract.

8. Build a frequency table for every distinct symbol

Counting only `R` needs one integer. Counting every symbol needs a mapping from symbol to count:

PYTHON
```
def symbol_frequencies(signal):
    """Return the occurrence count for every signal symbol."""
    frequencies = {}

    for symbol in signal:
        frequencies[symbol] = frequencies.get(symbol, 0) + 1

    return frequencies
```

Trace `"ABACA"`:

Symbol	Mapping before	Mapping after
`A`	`{}`	`{'A': 1}`
`B`	`{'A': 1}`	`{'A': 1, 'B': 1}`
`A`	`{'A': 1, 'B': 1}`	`{'A': 2, 'B': 1}`
`C`	`{'A': 2, 'B': 1}`	`{'A': 2, 'B': 1, 'C': 1}`
`A`	`{'A': 2, 'B': 1, 'C': 1}`	`{'A': 3, 'B': 1, 'C': 1}`

Check ordinary and empty behavior:

```python
assert symbol_frequencies("ABACA") == {"A": 3, "B": 1, "C": 1}
assert symbol_frequencies("") == {}
```

`.get(symbol, 0)` supplies the count before a symbol has appeared. The stored count then becomes state for later occurrences.

9. Group records when each key owns several values

Frequency counting stores one integer per key. Grouping stores a collection or other aggregate per key.

```python
def energies_by_tower(readings):
    """Return active energy values grouped by tower name."""
    groups = {}

    for reading in readings:
        if not reading["active"]:
            continue
        tower = reading["tower"]
        groups.setdefault(tower, []).append(reading["energy"])

    return groups
```

Use repeated tower names:

```python
repeated_readings = [
    {"tower": "north", "energy": 4, "active": True},
    {"tower": "south", "energy": 6, "active": True},
    {"tower": "north", "energy": 7, "active": True},
    {"tower": "south", "energy": 9, "active": False},
]

assert energies_by_tower(repeated_readings) == {
    "north": [4, 7],
    "south": [6],
}
```

Choose the stored value from the output contract:

- counts per tower → integer;
- energy values per tower → list;

- energy total per tower → running numeric total;
- latest reading per tower → one record.

All use a dictionary, but they are different algorithms because their updates and output shapes differ.

10. Remove duplicates without losing first-seen order

A set can remove duplicates, but a set result does not promise encounter order. When order matters, keep both a membership structure and an output structure:

```python
def unique_in_order(values):
    """Return first occurrences in their original order."""
    seen = set()
    result = []

    for value in values:
        if value not in seen:
            seen.add(value)
            result.append(value)

    return result
```

Check cases that expose the contract:

```python
assert unique_in_order(["R", "G", "R", "B"]) == ["R", "G", "B"]
assert unique_in_order([]) == []
assert unique_in_order(["R", "R", "R"]) == ["R"]
assert unique_in_order(["B", "R", "B"]) == ["B", "R"]
```

These structures have different jobs:

- `seen` answers “has this value appeared?”;
- `result` preserves “in what order did new values appear?”

Returning `list(set(values))` does not express the order promise, even when one small run happens to look right.

Checkpoint: selection, frequencies, grouping, and uniqueness

Choose state for selection and mappings

Identify the state and update that preserve each contract.

1. A running-best condition uses candidate_energy > best_energy. What happens on an equal-energy tie?

- **A.** The later item always replaces the earlier item.
- **B.** The current earlier item remains selected.
- **C.** Both items are removed.
- **D.** The loop returns None.

2. The result must keep every energy value under its tower name. What should each dictionary value contain?

- **A.** Only True or False
- **B.** The Python source line number
- **C.** A list of that tower’s energy values
- **D.** One unrelated tower name

3. Why does unique_in_order need both seen and result?

- **A.** seen supports membership while result preserves first-seen order.
- **B.** Every algorithm must create two collections.
- **C.** result raises errors and seen catches them.
- **D.** One stores code and the other runs it.

Write your choices before checking the answer key at the end of this book.

11. Compare neighbors without losing an endpoint

Some questions concern relationships between adjacent values: Did energy rise? Where did the symbol change? How many equal pairs occur?

An index-based scan names both positions:

```python
def rising_steps(values):
    """Return indexes where a value is greater than its predecessor."""
    rises = []

    for index in range(1, len(values)):
        if values[index] > values[index - 1]:
            rises.append(index)

    return rises
```

Check the first comparable index and small inputs:

```python
assert rising_steps([3, 5, 4, 8]) == [1, 3]
assert rising_steps([]) == []
assert rising_steps([7]) == []
assert rising_steps([7, 7]) == []
```

Starting at `1` is deliberate: index `0` has no predecessor. A paired traversal can express the values rather than indexes:

```python
values = [3, 5, 4, 8]
pairs = list(zip(values, values[1:]))
assert pairs == [(3, 5), (5, 4), (4, 8)]
```

Use indexes when the position itself is part of the result. Use paired values when only the relationship matters.

12. Scan a fixed-size window for a local pattern

A synchronization marker is the first consecutive window whose symbols are all distinct. Return the **1-based end position** of that window:

```python
def first_sync_end(signal, width):
    """Return the 1-based end of the first distinct window, or None."""
    if width <= 0:
        return None

    for end in range(width, len(signal) + 1):
        window = signal[end - width : end]
        if len(set(window)) == width:
            return end

    return None
```

Trace `"AABCDEF"` with width `4`:

end	Slice indexes	Window	Distinct?
4	0:4	AABC	no
5	1:5	ABCD	yes

The function returns `5`, meaning that five input symbols have been processed when the marker ends. It does not return the zero-based start index `1`.

Check boundaries:

```python
assert first_sync_end("AABCDEF", 4) == 5
assert first_sync_end("Z", 1) == 1
assert first_sync_end("ABCD", 4) == 4
assert first_sync_end("AAAA", 2) is None
assert first_sync_end("ABC", 4) is None
assert first_sync_end("ABC", 0) is None
```

Writing the slice and returned-position meanings beside the trace prevents a common off-by-one error.

13. Combine patterns only when the contract connects them

Suppose the task is: “From active readings, return normalized tower names.” A filter followed by a transform directly matches the words:

```python
active_names = [
    reading["tower"].upper()
    for reading in readings
    if reading["active"]
]

assert active_names == ["NORTH", "SOUTH"]
```

A longer sequence may be clearer as named stages:

```python
active = [reading for reading in readings if reading["active"]]
energies = [reading["energy"] for reading in active]
total = sum(energies)

assert total == 10
```

One loop can combine filter, transform, and accumulate:

```python
total = 0
for reading in readings:
    if reading["active"]:
        total += reading["energy"]

assert total == 10
```

Neither version is automatically superior. The staged form exposes reusable intermediate results. The combined form avoids building intermediate lists. Use the contract, scale, and clarity—not a slogan that fewer loops are always better.

14. Built-ins express patterns but do not decide the contract

Python provides concise operations:

```python
values = [4, 9, 6]

assert sum(values) == 19
assert min(values) == 4
assert max(values) == 9
assert any(value > 8 for value in values) is True
assert all(value > 0 for value in values) is True
assert sorted(values) == [4, 6, 9]
```

Before choosing one, identify the promised result:

- `sum` accumulates numeric values;
- `min` and `max` select one extreme value;
- `any` and `all` return Booleans and may stop early;
- `sorted` returns a complete ordered list.

A built-in cannot infer eligibility or a product-specific tie rule. Provide those decisions explicitly:

```python
best_active = max(
    (reading for reading in readings if reading["active"]),
    key=lambda reading: reading["energy"],
    default=None,
)

assert best_active["tower"] == "south"
```

This is concise only after the generator expression, `key`, and `default` are understood. A direct loop is preferable when it better exposes the current learning goal or complex tie behavior.

15. Repair patterns chosen from one keyword

“Find all red signals”

A first-match search is wrong because **all** matching values must appear in the result.

```python
red_positions = [
    index
    for index, symbol in enumerate("RGBRRY")
    if symbol == "R"
]

assert red_positions == [0, 3, 4]
```

“Return unique signals”

The word *unique* is ambiguous. It might mean distinct values in first-seen order:

```python
assert unique_in_order("ABACA") == ["A", "B", "C"]
```

Or it might mean values that occur exactly once:

```python
frequencies = symbol_frequencies("ABACA")
once_only = [symbol for symbol in "ABACA" if frequencies[symbol] == 1]
assert once_only == ["B", "C"]
```

“Choose the best reading”

Define eligibility, comparison direction, tie rule, no-result behavior, returned shape, and whether every item must be inspected. Only then choose running-best, `max`, sorting, or another strategy.

Checkpoint: neighbors, windows, combinations, and built-ins

Adapt and combine patterns

Choose the implementation decision that matches the promised result.

1. Why does an index-based adjacent-pair scan usually start at index 1?

- **A.** Index 0 is always invalid in Python.
- **B.** Lists automatically remove index 0.
- **C.** range cannot produce zero.
- **D.** Index 1 is the first value that has a predecessor.

2. first_sync_end("AABCDEF", 4) finds window "ABCD" at slice 1:5. Why does it return 5?

- **A.** Five is the number of processed symbols at the window’s 1-based end.
- **B.** Every width-four window returns five.
- **C.** The set contains five symbols.
- **D.** Python slices include index 5.

3. When may several named passes be preferable to one combined loop?

- **A.** Whenever the input is empty
- **B.** When intermediate results clarify responsibilities or are reused
- **C.** Because several passes always use less memory
- **D.** Because one loop cannot contain an if statement

Write your choices before checking the answer key at the end of this book.

16. Lab: decode the lantern relay

Use this signal and reading sequence:

```python
relay_signal = "RRGYBGRY"
relay_energy = [2, 5, 3, 8, 1, 4, 7, 2]
```

Implement these contracts:

```python
def count_symbols(signal):
    """Return a frequency dictionary for signal."""
    pass


def first_energy_at_least(values, threshold):
    """Return the first qualifying energy value, or None."""
    pass


def rising_energy_positions(values):
    """Return indexes whose values are greater than their predecessors."""
    pass


def first_distinct_window_end(signal, width):
    """Return the 1-based end of the first distinct window, or None."""
    pass
```

Use these progressive checks:

```python
assert count_symbols("") == {}
assert count_symbols("RRGYBGRY") == {"R": 3, "G": 2, "Y": 2, "B": 1}

assert first_energy_at_least([], 6) is None
assert first_energy_at_least(relay_energy, 6) == 8

assert rising_energy_positions([]) == []
assert rising_energy_positions([5]) == []
assert rising_energy_positions(relay_energy) == [1, 3, 5, 6]

assert first_distinct_window_end("AAAA", 2) is None
assert first_distinct_window_end("RRGYBGRY", 4) == 5
```

For each function, record:

- promised result shape;
- persistent state;
- stopping rule;

- empty/no-result value;
- order requirement; and
- one boundary that could expose a mistake.

Hint: solve the window on paper

For width four, list candidate windows in order. `RRGY` repeats `R`; the next window, `RGYB`, contains four distinct symbols and ends after five processed symbols.

Show the window evidence

The width-four windows begin `RRGY`, `RGYB`, `GYBG`, `YBGR`, and `BGRY`. `RGYB` is the first all-distinct window and ends after position 5, which agrees with the supplied assertion.

17. Name the state and stopping rule

Complete this table for your lab before checking the summary:

Function	State	Stops early?	No-result/empty output
`count_symbols`			
`first_energy_at_least`			
`rising_energy_positions`			
`first_distinct_window_end`			

Compare the pattern choices

- `count_symbols`: frequency dictionary; no early stop; `{}` for empty input.
- `first_energy_at_least`: no persistent collection, only current value; stops on a match; `None` for no match.
- `rising_energy_positions`: result list plus adjacent values/index; examines all pairs; `[]` for fewer than two values.
- `first_distinct_window_end`: current candidate window/end position; stops on a match; `None` for invalid width or no matching window.

Key points

- Start from the promised result shape, then choose state, update, stopping, order, tie, and no-result behavior.
- Transform, filter, count, accumulate, search, select, group, deduplicate, compare neighbors, and scan windows answer different questions.
- Built-ins express familiar algorithms but cannot decide product rules.
- Several clear passes can be better than one overloaded pass; one combined pass can be better when intermediate collections add no value.

- A pattern is only a starting structure. Acceptance examples determine whether its adaptation is correct.

References

- Python tutorial: looping techniques
- Python tutorial: list comprehensions
- Python documentation: `any`, `all`, `min`, `max`, `sum`, and other built-ins
- Python documentation: dictionary `get` and `setdefault`
- Python documentation: `enumerate` and `zip`

Know It Works and Understand the Cost

Online lesson and runnable notebook

Explain why a small algorithm is correct, show that it terminates, and compare alternatives through input size, operation growth, and extra-space trade-offs.

- **Level:** Python Foundations
- **Estimated time:** 4–5 hours
- **You will learn:** Use cases, counterexamples, invariants, termination, operation counts, and time/space growth to defend an algorithm choice.
- **Practice in:** Google Colab, JupyterLab, or a local editor

This lesson studies a trail-marker cleanup tool. Hikers record marker codes in encounter order, but some markers are repeated. The program must keep the first occurrence of each code and preserve order.

```python
markers = ["oak", "river", "oak", "ridge", "river"]
expected = ["oak", "river", "ridge"]
```

Keep these questions beside the notebook:

1. Which cases could disprove a confident claim that the algorithm works?
2. What remains true after each processed marker?
3. Which changing value guarantees that the loop eventually stops?
4. What does input size `n` count for this problem?
5. How does work change when `n` doubles?
6. What time, memory, order, and input-type promises differ among alternatives?

1. One passing trail does not cover every route

This implementation passes the opening sample:

```python
def remove_adjacent_repeats(values):
    """Return values that differ from the immediately previous value."""
    result = []

    for value in values:
        if not result or value != result[-1]:
            result.append(value)

    return result
```

Run it on adjacent duplicates:

```python
assert remove_adjacent_repeats(["oak", "oak", "river"]) == ["oak", "river"]
```

But the actual contract says remove **every later repeat**, not only adjacent repeats:

```python
actual = remove_adjacent_repeats(["oak", "river", "oak"])
assert actual == ["oak", "river", "oak"]
```

The code does what its docstring says. It does not meet the trail cleanup contract. A program can be internally consistent and still solve the wrong problem.

The small input `['oak', 'river', 'oak']` is a counterexample to the claim that neighbor comparison is sufficient.

2. Give every case a reason to exist

Build a behavior table before repairing the code:

Case	Input	Expected	Rule exposed
empty	`[]`	`[]`	output shape for no values
one marker	`['oak']`	`['oak']`	smallest successful input
all repeated	`['oak', 'oak', 'oak']`	`['oak']`	repeated-value removal
all distinct	`['oak', 'river']`	`['oak', 'river']`	preserve every new value
separated repeat	`['oak', 'river', 'oak']`	`['oak', 'river']`	duplicates need not be adjacent
alternating	`['oak', 'river', 'oak', 'river']`	`['oak', 'river']`	repeated lookup over time
order-sensitive	`['ridge', 'oak', 'ridge']`	`['ridge', 'oak']`	preserve first-seen order

Represent these cases with plain data and assertions:

```python
cleanup_cases = [
    ([], []),
    (["oak"], ["oak"]),
    (["oak", "oak", "oak"], ["oak"]),
    (["oak", "river"], ["oak", "river"]),
    (["oak", "river", "oak"], ["oak", "river"]),
    (["oak", "river", "oak", "river"], ["oak", "river"]),
    (["ridge", "oak", "ridge"], ["ridge", "oak"]),
]
```

Unit 13 will organize cases with pytest and parametrization. Here, the important skill is selecting cases that challenge distinct rules.

3. Frame the claim with preconditions and postconditions

A **precondition** describes supported input before the algorithm begins. A **postcondition** describes what must be true when it finishes.

For a set-assisted cleanup algorithm:

- **Precondition:** `values` is a finite iterable of hashable values.
- **Postcondition 1:** the result contains each distinct input value once.
- **Postcondition 2:** result order follows each value's first input occurrence.
- **Postcondition 3:** the input is unchanged.

Hashability is part of the precondition because a set needs hashable elements. The function does not need to catch or recover from unsupported values in this unit; Unit 8 teaches exception boundaries.

A separate algorithm could support unhashable values such as lists by using list membership. The algorithms therefore do not have identical input contracts. That difference belongs in any comparison.

4. Shrink a failure until one rule is visible

Suppose a long trail fails:

```python
long_failure = [
    "oak",
    "river",
    "ridge",
    "field",
    "oak",
    "bridge",
    "river",
]
```

Remove irrelevant values while keeping the failure. If the adjacent-only algorithm still fails on this input, shrink again:

```python
smaller_failure = ["oak", "river", "oak"]
```

Now one relationship is visible: the repeated `oak` is separated from its first occurrence. This minimal counterexample suggests that the algorithm must remember more than the immediately previous value.

A counterexample is a design instrument

Do not collect a large failing input only as proof that something went wrong. Reduce it until the missing state or rule becomes obvious enough to guide the next algorithm. :::

5. State what is true for the processed prefix

Use the stable-order implementation from Lesson 3:

```python
def unique_in_order(values):
    """Return first occurrences in encounter order."""
    seen = set()
```

PYTHON (CONTINUED)

```
    result = []

    for value in values:
        if value not in seen:
            seen.add(value)
            result.append(value)

    return result
```

An **invariant** is a statement that remains true at a chosen point during the loop. Immediately after each iteration:

`result` contains the first occurrence of every distinct value in the processed prefix, in encounter order, and `seen` contains exactly the same values.

Trace the opening input:

Processed prefix	seen	result	Invariant true?
`[]`	`set()`	`[]`	yes
`['oak']`	`{'oak'}`	`['oak']`	yes
`['oak', 'river']`	`{'oak', 'river'}`	`['oak', 'river']`	yes
`['oak', 'river', 'oak']`	unchanged	unchanged	yes
`['oak', 'river', 'oak', 'ridge']`	adds `ridge`	appends `ridge`	yes

Why this statement supports correctness:

1. **Before the loop**, no values are processed; both structures are empty, so the statement is true.
2. **For a new value**, adding it to both structures preserves membership and order.
3. **For a repeated value**, changing neither structure preserves the first occurrence only.
4. **After the loop**, the processed prefix is the entire input, so the invariant becomes the promised result.

This is a correctness argument in ordinary language, not formal proof notation.

Checkpoint: cases, contracts, and invariants

Build a correctness claim

Choose the evidence that directly supports the cleanup contract.

1. Which smallest input disproves an algorithm that removes only adjacent duplicates?

- **A.** ["oak"]
- **B.** ["oak", "river", "oak"]
- **C.** []
- **D.** ["oak", "oak"]

2. Why does the set-assisted version state that values must be hashable?

- **A.** Hashing automatically sorts the result.
- **B.** Hashable values are always strings.
- **C.** The result list cannot hold integers.
- **D.** Set membership requires hashable elements.

3. Why does the processed-prefix invariant describe the final result after the loop?

- **A.** The processed prefix has become the entire finite input.
- **B.** Every invariant sorts its input.
- **C.** The set is converted into a traceback.
- **D.** The loop stops before reading any value.

Write your choices before checking the answer key at the end of this book.

6. Explain why the loop eventually stops

Correct partial results are not enough if the algorithm can run forever. `unique_in_order` uses a `for` loop over a finite input. Each iteration consumes one next value, so the number of unprocessed values decreases by one. Eventually none remain.

A `while` version makes the progress variable visible:

```python
def unique_in_order_by_index(values):
    """Return first occurrences in encounter order."""
    seen = set()
    result = []
    index = 0

    while index < len(values):
        value = values[index]
        if value not in seen:
            seen.add(value)
            result.append(value)
        index += 1

    return result
```

`index += 1` is essential. Without it, the condition remains true for any non-empty input and the same value is processed repeatedly.

For this loop, a **progress measure** is `len(values) - index`. It begins as a non-negative integer, decreases by one, and cannot decrease forever without reaching zero.

You do not need that expression for every ordinary `for` loop. Use it when the stopping argument is unclear, especially for `while`, nested state, or a manually updated position.

7. Preserve behavior while changing an implementation

Complete the set-assisted function, then run every case:

```python
for values, expected in cleanup_cases:
    assert unique_in_order(values) == expected
```

Check the ownership promise separately:

```python
source = ["oak", "river", "oak"]
before = source.copy()
result = unique_in_order(source)

assert result == ["oak", "river"]
assert source == before
assert result is not source
```

Value equality checks behavior; identity checks that the function returned a separate list. Neither check replaces the other.

If you later change the algorithm, keep these checks unchanged. A faster result that reorders markers or mutates the input does not preserve the same contract.

8. Define the input size before discussing speed

Use `n` for the number of input markers:

```python
values = ["oak", "river", "oak", "ridge"]
n = len(values)
assert n == 4
```

For other problems, `n` could mean characters, records, rows, or vertices. State it. A grid may need two sizes—rows `r` and columns `c`—rather than hiding both inside `n`.

Wall-clock measurements of tiny inputs vary with the computer and background work. Begin by counting an operation whose repetition explains the algorithm. For cleanup, membership checks are important.

```python
def unique_with_membership_count(values):
    seen = set()
    result = []
    membership_checks = 0

    for value in values:
        membership_checks += 1
        if value not in seen:
            seen.add(value)
            result.append(value)

    return result, membership_checks
```

Each input value causes one set-membership check:

```python
assert unique_with_membership_count([])[1] == 0
assert unique_with_membership_count(["a"])[1] == 1
assert unique_with_membership_count(["a", "b"])[1] == 2
assert unique_with_membership_count(["a", "b", "c", "d"])[1] == 4
```

Doubling `n` doubles the number of membership checks in this loop.

9. Describe growth before naming its category

Consider four common shapes:

One lookup independent of collection traversal

```python
first = values[0]
```

For a non-empty list, retrieving index `0` does not scan all values. We call this **constant growth**, written `O(1)`.

Repeatedly halve the remaining search range

If a sorted search space is cut roughly in half at each step, doubling its size adds about one additional step. This is **logarithmic growth**, written `O(log n)`. Binary search is a familiar example; its full implementation is not required in this unit.

Inspect every value once

```python
for value in values:
    print(value)
```

The number of iterations grows with `n`. This is **linear growth**, `O(n)`.

Compare many pairs

```python
for left_index in range(len(values)):
    for right_index in range(left_index + 1, len(values)):
        print(values[left_index], values[right_index])
```

For an all-pairs comparison, doubled input approaches four times as many pairs. This is **quadratic growth**, $O(n^2)$.

Sorting comparison-based data belongs to another important category, `O(n log n)`. You already use `sorted`; implementing sorting algorithms is outside this Foundations unit.

Big-O notation describes how growth behaves as input becomes large. It does not state exact seconds, and it intentionally ignores fixed multipliers and lower-order terms.

10. Best and worst cases can differ

A first-match search may stop immediately or inspect the whole input:

```python
def index_of(values, target):
    """Return the first target index, or None."""
    for index, value in enumerate(values):
        if value == target:
            return index
    return None
```

Compare positions:

```python
values = ["a", "b", "c", "d"]

assert index_of(values, "a") == 0
assert index_of(values, "d") == 3
assert index_of(values, "z") is None
```

- **Best case:** the target is first; one comparison.
- **Worst case:** the target is last or absent; `n` comparisons.
- **Ordinary observed case:** depends on real input distribution.

When the function must support any listed input, worst-case growth is a useful shared comparison. It does not predict how often each case occurs in a specific product.

Checkpoint: input size and growth

Reason about growing inputs

Identify what grows and why before choosing a complexity label.

1. A loop performs one membership check per input marker. What happens to that count when n doubles?

- **A.** It stays exactly one.
- **B.** It becomes half as large.
- **C.** It roughly doubles.
- **D.** It automatically becomes quadratic.

2. What is a worst case for first-match search?

- **A.** The target is last or absent.
- **B.** The target is first.
- **C.** The input contains one item that matches.
- **D.** The function name is short.

3. What does $O(n^2)$ communicate?

- **A.** The program always takes exactly n^2 seconds.
- **B.** The algorithm cannot be correct.
- **C.** The input must contain square numbers.
- **D.** The important operation can grow proportionally to the square of input size.

Write your choices before checking the answer key at the end of this book.

11. A nested duplicate scan repeats earlier work

A baseline algorithm can support equality-comparable values even when they are unhashable:

```python
def unique_with_list_membership(values):
    """Return first occurrences using result-list membership."""
    result = []

    for value in values:
        if value not in result:
            result.append(value)

    return result
```

It is correct for the stated order contract:

```python
assert unique_with_list_membership(["a", "b", "a"]) == ["a", "b"]
assert unique_with_list_membership([[1], [2], [1]]) == [[1], [2]]
```

But `value not in result` can scan the growing result list. On all-distinct input, approximate equality comparisons grow like this:

n	Earlier result items inspected across iterations
1	0
2	1
4	6
8	28
16	120

The sum is `0 + 1 + 2 + ... + (n - 1)`. Doubling `n` makes the count approach four times as large. This worst case is $O(n^2)$.

“Only one visible `for` loop” does not imply linear time. The membership operation inside it performs its own work.

12. A set trades extra memory for faster ordinary membership

The set-assisted version performs one ordinary set lookup per item:

```python
def unique_with_set(values):
    """Return first occurrences of hashable values in encounter order."""
    seen = set()
    result = []

    for value in values:
        if value not in seen:
            seen.add(value)
            result.append(value)

    return result
```

For ordinary set behavior, lookup and insertion are treated as average `O(1)`, so the whole traversal has average `O(n)` time. The function also stores up to `n` values in `seen` in addition to the result, so its extra working space is `O(n)`.

Be precise about the claim:

- it is an ordinary/average hash-table cost, not a promise that every conceivable lookup under every collision pattern takes identical time;
- it requires hashable values;
- it preserves order because `result`, not the set, is returned;
- it does not mutate the input.

The list-membership version supports a wider class of equality-comparable values and uses no separate membership collection, but its worst-case time grows quadratically. The “better” choice depends on the input contract and scale.

13. Sorting, scanning, and hashing preserve different promises

A sorted-copy strategy can group equal values together:

```python
def unique_sorted(values):
    """Return distinct values in sorted order."""
    ordered = sorted(values)
    result = []

    for value in ordered:
        if not result or value != result[-1]:
            result.append(value)

    return result
```

It returns a different order:

```python
values = ["ridge", "oak", "ridge", "field"]
assert unique_sorted(values) == ["field", "oak", "ridge"]
assert unique_with_set(values) == ["ridge", "oak", "field"]
```

Compare the contracts before comparing speed:

Strategy	Order returned	Supported values	Typical/worst growth discussed here	Extra working space
result-list membership	first-seen	equality-comparable	worst `O(n²)`	result only
set-assisted	first-seen	hashable	average `O(n)`	set plus result
sorted copy + adjacent scan	sorted	mutually orderable	`O(n log n)` for sort plus scan	sorted copy plus result

The sorted version is not a faster replacement for a first-seen-order contract. It solves a different output-order problem.

14. Big-O does not choose the program for you

Suppose a configuration contains at most six marker codes and accepts small nested lists. The result-list version may be the clearest supported solution. Its theoretical quadratic worst case is unlikely to matter at that scale.

Suppose a stream contains millions of hashable identifiers and order must be preserved. The set-assisted version avoids repeated scans and has a compelling trade-off.

Ask:

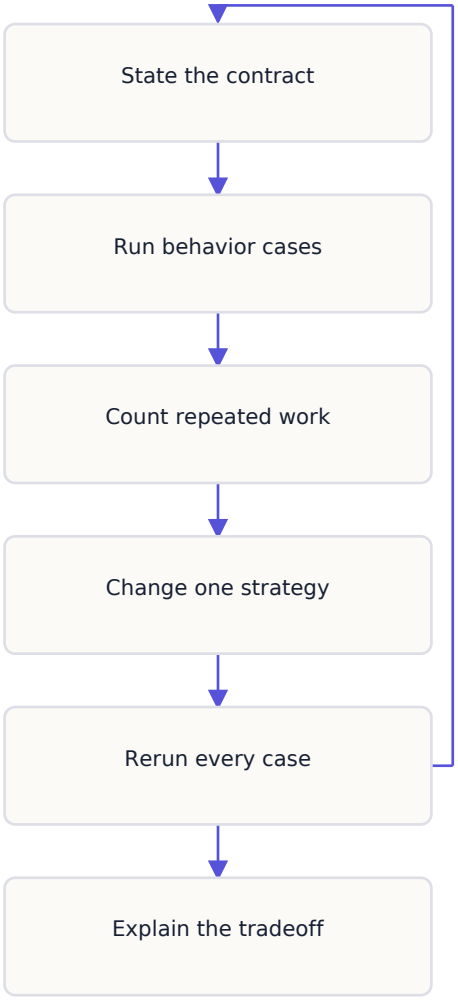
- What scale is actually supported?
- Is the code on a frequent path?
- Which input types must work?
- Is first-seen order required?
- May the program use additional memory?
- Is the simpler alternative already comfortably within the constraint?

Measured performance is useful after these questions. The standard-library `timeit` module and profiling tools appear naturally in later project/tooling work. A tiny timing run should not replace a correctness suite or a growth explanation.

15. Improve only after behavior is protected

Use this sequence:

The improvement cycle preserves the same contract while replacing one source of repeated work.



Rerun every case → State the contract: Behavior changed

For trail cleanup:

1. freeze the first-seen-order, non-mutation contract;
2. keep all acceptance cases unchanged;
3. identify repeated result-list membership scanning;
4. add a `seen` set for hashable inputs;
5. rerun value, order, empty, repeat, and ownership checks;
6. document average linear time and linear extra working space; and
7. retain the baseline if unhashable values remain a supported requirement.

An optimization is complete only when both behavior and trade-offs are explicit.

Checkpoint: compare complete contracts

Choose a defensible trade-off

Compare behavior, supported input, time growth, and extra space together.

1. Why can one visible loop with "value not in result" take quadratic time on all-distinct input?

- **A.** append sorts the input every time.
- **B.** Python runs every loop twice.
- **C.** The result list becomes immutable.
- **D.** Each list-membership check may scan a growing result.

2. What does the seen set add to the stable cleanup algorithm?

- **A.** Average fast membership at the cost of extra space and a hashable-value precondition
- **B.** Automatic alphabetical result order with no memory
- **C.** Support for every unhashable value
- **D.** A guarantee that no equality checks occur

3. Why is unique_sorted not a drop-in replacement for first-seen stable cleanup?

- **A.** sorted cannot process strings.
- **B.** A for loop cannot follow sorted.
- **C.** It changes the promised result order.
- **D.** It always mutates the caller’s list.

Write your choices before checking the answer key at the end of this book.

16. Lab: compare campsite-conflict detectors

A campsite request contains site IDs. A conflict exists when an ID appears more than once.

```python
requests = ["C12", "A03", "C12", "B07"]
```

Implement and compare three functions:

```python
def has_conflict_nested(site_ids):
    """Return True when any two positions contain the same ID."""
    pass


def has_conflict_seen(site_ids):
    """Return True when a hashable ID has appeared before."""
    pass


def has_conflict_sorted(site_ids):
    """Return True when adjacent values in a sorted copy are equal."""
    pass
```

Behavior checks

```python
conflict_cases = [
    ([], False),
    (["A03"], False),
    (["A03", "A03"], True),
    (["A03", "B07", "A03"], True),
    (["A03", "B07", "C12"], False),
]

for site_ids, expected in conflict_cases:
    assert has_conflict_nested(site_ids) is expected
    assert has_conflict_seen(site_ids) is expected
    assert has_conflict_sorted(site_ids) is expected
```

Ownership checks

```python
source = ["C12", "A03", "B07"]
before = source.copy()

has_conflict_nested(source)
has_conflict_seen(source)
has_conflict_sorted(source)

assert source == before
```

Reasoning tasks

1. State the precondition for each function.
2. Write an invariant for the `seen` version.
3. Explain why each loop terminates.
4. Add counters for equality or membership checks on all-distinct inputs of length 4, 8, and 16.
5. Classify worst-case time growth and extra working space.
6. Recommend a version for at most six possibly unhashable IDs.
7. Recommend a version for one million hashable string IDs where order is not part of the Boolean result.

Hint: nested pairs need two indexes

For each left index, compare only positions to its right. Return `True` on the first equal pair and `False` after every pair has been considered.

Show a comparison strategy after attempting the lab

The nested version uses two index ranges and no extra membership structure; it supports equality-comparable values but has quadratic worst-case comparisons. The seen version returns on the first repeated ID and has average linear time with linear extra set space for hashable IDs. The sorted-copy version sorts without mutating the input, then checks adjacent pairs; it needs mutually orderable values and `O(n log n)` sorting time. For six unhashable values, the nested version may be simplest. For one million hashable strings, the seen version is usually the strongest starting choice.

17. Defend the choice in plain language

Finish the lab with six sentences:

1. **Contract:** what behavior and input types the chosen function supports.
2. **Cases:** which input most strongly challenges the implementation.
3. **Invariant:** what remains true after each completed iteration.
4. **Termination:** what progresses toward the stopping condition.
5. **Cost:** how time and extra working space grow with `n`.
6. **Trade-off:** why one rejected alternative is less suitable for this contract and scale.

This explanation is part of the result. It allows a future maintainer to decide whether a changed requirement invalidates the choice.

Key points

- Passing one familiar example does not establish correctness; purposeful boundaries and counterexamples challenge distinct rules.
- Preconditions frame supported input, postconditions frame the result, and a loop invariant connects partial work to the finished promise.
- A termination argument identifies progress through a finite problem.
- Define input size and count important repeated operations before naming a growth category.
- Compare time, extra space, supported values, ordering, and mutation behavior together.
- Improve an algorithm only while preserving the same behavior suite—or state clearly that the contract changed.

References

- Python reference: the `assert` statement
- Python documentation: set and frozenset operations
- Python documentation: sequence membership operations
- Python documentation: `sorted` and its returned list
- Python sorting guide: keys and sort stability

Unit Challenge: Wake the Clockwork Observatory

Online lesson and runnable notebook

Derive an observatory wake-up code by decomposing a signal puzzle into frequency, selection, window-search, and running-total algorithms.

- **Level:** Python Foundations · Unit 7 challenge
- **Estimated time:** 60–120 minutes
- **Outcome assessed:** Specify, decompose, implement, and defend several cooperating algorithms.
- **Evidence:** Passing progressive assertions, a wake-up code, cost notes, a clean rerun, and one debugging record

1. Wake the observatory with a code you derive

The hilltop clockwork observatory has slept through three meteor showers. Its brass dome will move only when a signal calibrator produces a three-part code:

```text
<sync position>-<power step>-<anchor symbol>
```

You receive a symbol signal and matching energy readings:

```python
signal = "AABCDEFAC"
energy_readings = [2, 1, 3, 2, 4, 1, 5, 2, 1]
window_width = 4
power_target = 10
```

Your program must discover:

1. where the first all-distinct symbol window ends;
2. when accumulated energy first reaches the target; and
3. which symbol occurs most often, with first appearance winning a tie.

For the supplied data, the finished program should reveal:

```text
Observatory awake: 5-5-A
```

You will choose the state and conditions that produce that result. The page supplies names, docstrings, staged assertions, three hints, and a hidden solution for use after a serious attempt.

Work independently, but not without support

Plan for one or two focused hours. Read one contract, make the matching small check pass, and rerun earlier checks before moving forward. Open a hint only after writing down the exact value or condition that blocks you. :::

2. Read the signal rules and acceptance examples

Frequency and anchor rules

- Count every symbol exactly as written.
- The anchor is the symbol with the greatest count.
- If counts tie, the symbol that appears earlier in the signal wins.
- An empty signal has no anchor, represented by `None`.

```python
assert {"A": 3, "B": 1, "C": 2, "D": 1, "E": 1, "F": 1} == {
    "A": 3,
    "B": 1,
    "C": 2,
    "D": 1,
    "E": 1,
    "F": 1,
}

# A and B both occur twice; A appears first.
tie_signal = "ABBA"
tie_anchor = "A"
```

The first assertion only confirms the hand-counted mapping is well formed. Your function checks begin in Section 5.

Synchronization-window rules

A window is a consecutive slice with exactly `window_width` symbols. It is synchronized when every symbol inside it is distinct. Return the **1-based end position**, which is also the number of symbols processed when the window ends. Return `None` if no such window exists.

For `"AABCDEFAC"` and width `4`:

End position	Window	All distinct?
4	`AABC`	no
5	`ABCD`	yes

The synchronization result is therefore `5`.

Power rules

Add readings from left to right. Return the **1-based step** where the running total first becomes greater than or equal to the target. Return `None` if all readings are consumed without reaching it.

Step	Reading	Running total	Reached 10?
1	2	2	no
2	1	3	no
3	3	6	no
4	2	8	no

Step	Reading	Running total	Reached 10?
5	4	12	yes

The power result is `5`. Stop there because the contract asks for the first reached step.

Input and output constraints

1. `signal` is a finite string.
2. `readings` is a finite list of non-negative numbers.
3. `width` and `target` are positive integers.
4. Functions must not modify supplied inputs.
5. Use the supplied names and returned shapes.
6. Do not hard-code `5`, `5`, `A`, or the final code.
7. If the anchor, synchronization, or power result is missing, the complete code is `None`.
8. Explain time and extra-space growth in plain language after the checks pass.

Before implementation, add one acceptance example of your own for each:

- a frequency tie;
- a window that never synchronizes; and
- an energy sequence that never reaches its target.

3. Start from the contract

Run this cell unchanged. Replace each `pass` while preserving the public names and docstrings.

PYTHON
```
def symbol_frequencies(signal):
    """Return a dictionary that counts every symbol in signal."""
    pass


def choose_anchor(signal, frequencies):
    """Return the most frequent symbol; the first appearance wins a tie."""
    pass


def first_sync_end(signal, width):
    """Return the 1-based end of the first all-distinct window, or None."""
    pass


def first_power_step(readings, target):
    """Return the 1-based step where the running total reaches target, or None."""
    pass


def build_wakeup_code(signal, readings, width, target):
    """Return '<sync>-<power>-<anchor>', or None if calibration cannot finish."""
    pass
```

Use this responsibility table before writing loops:

Function	Result shape	State to consider	May stop early?
`symbol_frequencies`	dictionary	counts by symbol	no
`choose_anchor`	symbol or `None`	best symbol/count	yes only for empty input
`first_sync_end`	integer or `None`	candidate window/end	yes on first match
`first_power_step`	integer or `None`	running total/step	yes on reached target
`build_wakeup_code`	string or `None`	returned helper results	after either failed result

You may implement `choose_anchor` by traversing `signal` and consulting the frequency mapping. That makes encounter-order tie behavior visible without sorting the dictionary.

4. Calibrate one mechanism at a time

Stage A: count symbols

Start with empty and repeated inputs. Do not work on anchor selection until both frequency checks pass.

A useful trace for `"ABACA"` has columns for current symbol and mapping after the update.

Stage B: select the anchor

Keep the current anchor when a candidate’s count is equal. Replace only when the candidate count is strictly greater. Repeated visits to the same symbol should not change the result.

Stage C: find synchronization

Name the meaning of `end` before coding. For an end-exclusive Python slice, `signal[end - width:end]`, the same `end` is already the promised 1-based count of processed symbols.

Stage D: accumulate power

Use `enumerate(readings, start=1)` or convert a zero-based index deliberately. Add the current reading before asking whether the target has been reached.

Stage E: assemble the code

Call the four helpers. If the sync result, power result, or anchor is `None`, return `None`. Otherwise return one formatted string.

After every stage:

1. run its smallest assertion;
2. trace actual state if it fails;
3. change one condition or update;
4. rerun the stage; and
5. rerun all earlier stages.

5. Run progressive assertions

Do not change these expected values merely to make a check pass. If code and expectation disagree, compare both with the written rules and hand traces.

Symbol-frequency evidence

PYTHON

```
assert symbol_frequencies("") == {}
assert symbol_frequencies("A") == {"A": 1}
assert symbol_frequencies("ABACA") == {"A": 3, "B": 1, "C": 1}
assert symbol_frequencies(signal) == {
    "A": 3,
    "B": 1,
    "C": 2,
    "D": 1,
    "E": 1,
    "F": 1,
}
```

Anchor and tie evidence

PYTHON

```
assert choose_anchor("", {}) is None
assert choose_anchor("A", {"A": 1}) == "A"
assert choose_anchor("ABACA", {"A": 3, "B": 1, "C": 1}) == "A"
assert choose_anchor("ABBA", {"A": 2, "B": 2}) == "A"
assert choose_anchor("BAAB", {"B": 2, "A": 2}) == "B"
```

Synchronization evidence

PYTHON

```
assert first_sync_end("Z", 1) == 1
assert first_sync_end("ABCD", 4) == 4
assert first_sync_end("AABCDEF", 4) == 5
assert first_sync_end("AAAA", 2) is None
assert first_sync_end("ABC", 4) is None
assert first_sync_end(signal, window_width) == 5
```

Running-power evidence

PYTHON

```
assert first_power_step([5], 5) == 1
assert first_power_step([2, 3, 1], 5) == 2
assert first_power_step([2, 3, 1], 7) is None
assert first_power_step([], 1) is None
assert first_power_step(energy_readings, power_target) == 5
```

Complete wake-up evidence

PYTHON

```
assert build_wakeup_code("ABCD", [2, 3], 4, 5) == "4-2-A"
assert build_wakeup_code("AAAA", [2, 3], 2, 5) is None
assert build_wakeup_code("ABCD", [1, 1], 4, 5) is None
assert build_wakeup_code(signal, energy_readings, window_width, power_target) == (
    "5-5-A"
)
```

Ownership and changed-input evidence

```python
readings_before = energy_readings.copy()
signal_before = signal

code = build_wakeup_code(signal, energy_readings, window_width, power_target)

assert code == "5-5-A"
assert energy_readings == readings_before
assert signal == signal_before
```

Add one complete assertion with a new signal, new readings, a different width, and a different target. Calculate its expected code on paper first.

6. Use the hint ladder only when needed

Hint 1

Match each helper to a pattern: frequency table, running-best selection, fixed-size window search, running-total search, and function composition.

Hint 2

For frequencies, update `frequencies.get(symbol, 0) + 1`. For the anchor, scan the original signal and replace only when a symbol's count is strictly greater than the current anchor's count. For synchronization, compare `len(set(window))` with `width`. For power, add before checking the target.

Hint 3

Use these incomplete shapes:

```python
for symbol in signal:
    frequencies[symbol] = ...

for end in range(width, len(signal) + 1):
    window = signal[end - width : end]
    if ...:
        return end

for step, reading in enumerate(readings, start=1):
    total += reading
    if ...:
        return step
```

Every search also needs a return after the loop for the no-result path.

7. Keep debugging evidence

Keep one failure that helped you understand the puzzle. Record facts before changing code.

Failure	Actual evidence	Single-cause hypothesis	Controlled change	Verified rerun
What failed?	Exact returned value, state, or message	Which one condition/update fits?	What one thing changed?	Which focused and earlier checks pass?

Useful failures include:

- returning a zero-based start instead of a 1-based end;
- replacing the anchor on an equal frequency and therefore choosing the last tied symbol;
- checking power before adding the current reading;
- returning `None` from inside a loop after the first non-match; or
- constructing a code containing the text `None` instead of returning `None`.

Restart the notebook or Python process and run every cell from the contract through the complete assertions. A result that depends on hidden state is not a finished solution.

8. Compare with a complete solution

Show the complete implementation after attempting every stage

PYTHON

```
def symbol_frequencies(signal):
    """Return a dictionary that counts every symbol in signal."""
    frequencies = {}
    for symbol in signal:
        frequencies[symbol] = frequencies.get(symbol, 0) + 1
    return frequencies


def choose_anchor(signal, frequencies):
    """Return the most frequent symbol; the first appearance wins a tie."""
    anchor = None
    for symbol in signal:
        if anchor is None or frequencies[symbol] > frequencies[anchor]:
            anchor = symbol
    return anchor


def first_sync_end(signal, width):
    """Return the 1-based end of the first all-distinct window, or None."""
    for end in range(width, len(signal) + 1):
        window = signal[end - width : end]
        if len(set(window)) == width:
            return end
    return None


def first_power_step(readings, target):
    """Return the 1-based step where the running total reaches target, or None."""
    total = 0
    for step, reading in enumerate(readings, start=1):
        total += reading
        if total >= target:
            return step
    return None


def build_wakeup_code(signal, readings, width, target):
    """Return '<sync>-<power>-<anchor>', or None if calibration cannot finish."""
    frequencies = symbol_frequencies(signal)
    anchor = choose_anchor(signal, frequencies)
    sync_end = first_sync_end(signal, width)
    power_step = first_power_step(readings, target)

    if anchor is None or sync_end is None or power_step is None:
        return None
    return f"{sync_end}-{power_step}-{anchor}"
```

Why the main decisions work:

- `choose_anchor` scans in encounter order and replaces only for a strictly larger count, so an equal count keeps the earlier symbol.
- `end` begins at `width`; `signal[end - width:end]` has the requested width, and `end` is already the 1-based number of processed symbols.
- `first_power_step` adds the current reading before comparing with the target, so the returned step includes the reading that reaches it.
- The composition function returns `None` rather than formatting an incomplete code.

For a signal of length `n`, frequency counting and anchor selection each grow linearly. With fixed window width `w`, the window search checks up to about `n` windows and builds a set of up to `w` symbols for each, so its time is `O(n × w)` and its temporary window space is `O(w)`. Power search is linear in its number of readings in the worst case. The frequency dictionary uses space proportional to the number of distinct symbols.

9. Predict a changed calibration rule

The observatory engineer proposes two changes:

1 on an anchor-frequency tie, choose the **later** symbol; and

2 return the **zero-based window start** instead of the 1-based end.

Before editing code, answer:

- Which acceptance examples change?
- Which comparison changes from strict to inclusive, and why can repeated visits complicate that shortcut?
- How is a start index calculated from the current `end` and `width`?
- Does the wake-up code for the original input change?
- Which docstrings and cost statements remain valid?

A robust later-wins anchor strategy should compare distinct candidates or track the last position deliberately; blindly replacing on every equal count also replaces an anchor when the loop sees the **same** symbol again. Write a small tie trace before implementing the changed rule.

10. Check your understanding

Clockwork observatory calibration check

Trace the supplied contract before choosing an answer.

1. Why does choose_anchor replace only when a symbol’s count is strictly greater?

- **A.** Strict comparison alphabetizes the signal.
- **B.** Equal counts make every symbol invalid.
- **C.** An equal count must keep the earlier selected symbol.
- **D.** A dictionary cannot contain equal values.

2. For signal "AABCDEF" and width 4, window "ABCD" uses slice 1:5. What does first_sync_end return?

- **A.** 5
- **B.** 1
- **C.** 4
- **D.** 6

3. What should build_wakeup_code return when no distinct window exists?

- **A.** A code containing the word missing
- **B.** An empty dictionary
- **C.** The last window
- **D.** None

4. When should first_power_step compare the running total with the target?

- **A.** Before reading any values and never again
- **B.** After adding the current reading
- **C.** Only after all readings are consumed
- **D.** Before adding the current reading

5. Why can the straightforward window scan take work proportional to n times width?

- **A.** It sorts the entire signal for every symbol.
- **B.** It recursively calls every helper.
- **C.** It may build and inspect a width-sized set for many candidate windows.
- **D.** Every string operation is quadratic.

Write your choices before checking the answer key at the end of this book.

11. Decide whether the challenge is complete

Evidence rubric

Evidence	Ready to record when
Behavior	All supplied assertions and one learner-created case pass unchanged.
Decomposition	Each public function has one named responsibility and returned shape.
Reasoning	Tie, index, stopping, no-result, time, and extra-space decisions can be explained.
Debugging	One failure record connects exact evidence to one change and verified reruns.
Reproducibility	The complete observatory program works from a clean notebook state.

- ☐ Every supplied assertion passes without edited expected values.

- ☐ I added and hand-calculated one complete changed-input case.
- ☐ The inputs remain unchanged.
- ☐ I can explain the anchor tie, sync indexes, power stopping, and costs.
- ☐ I kept one debugging record and reran everything from a clean state.

This button stores a self-reported marker only in this browser. It does not submit or grade the program.

Key points

- The observatory puzzle becomes manageable when counting, selection, window search, accumulation, and composition have separate contracts.
- Progressive assertions reveal which responsibility is wrong without revealing the implementation.
- Tie rules, index meanings, stopping behavior, and no-result values belong in examples before they become conditions.
- Correct behavior, a clean rerun, and a plain-language cost explanation are all part of the finished solution.

Unit 8: Errors, Exceptions, and Debugging

Online lesson and runnable notebook

Classify failures, read tracebacks, handle expected exceptions, and repair bugs through small evidence-backed experiments.

- **Level:** Python Foundations · Unit 8
- **Estimated time:** 18–23 hours including the challenge
- **Unit outcome:** Classify syntax failures, runtime exceptions, and wrong results; read tracebacks across function calls; raise and handle only anticipated exceptions with useful context; and reproduce, inspect, isolate, repair, and verify a bug one evidence-backed hypothesis at a time.
- **Practice in:** Generated Colab notebooks and a local Python file when using a debugger

1. Welcome to the arcade repair desk

An arcade machine can fail in at least three noticeably different ways:

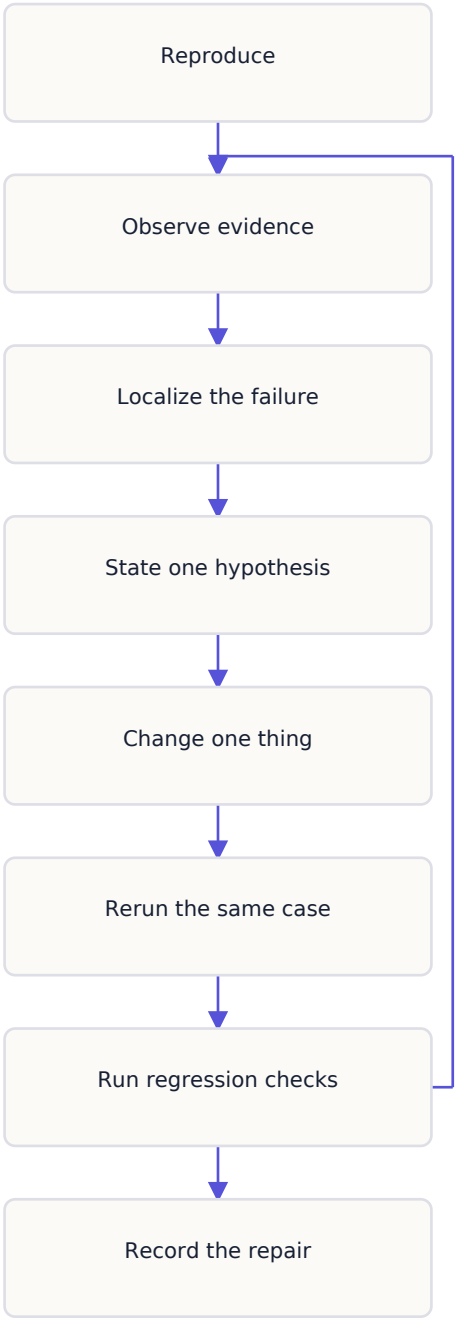
- its game script may not start because Python cannot understand the code;
- it may start and then stop when a player enters an unexpected command; or
- it may finish without an error while moving the character in the wrong direction.

Those are not variations of one problem. They leave different evidence and call for different first moves. This unit teaches you to treat a failure as an investigation rather than a guessing contest.

By the end, red tracebacks should feel like structured reports. A silent wrong answer should prompt you to write down the expected result and find the first place where state diverges. An anticipated bad input should lead to one narrow, useful exception boundary—not a blanket `try` block that hides every defect.

2. Follow the evidence loop

The investigation repeats until the repaired behavior survives both the original case and nearby regression checks.



Run regression checks → Observe evidence: still wrong

Run regression checks → Record the repair: stable

This loop is intentionally slower than random editing for the first few minutes. It is usually much faster over the whole investigation.

An evidence log makes the loop visible:

Reproduction	Observed evidence	Hypothesis	One change	Verification
`move_pixel((0, 0), ("LEFT", 1))`	returns `(1, 0)`	the left branch adds instead of subtracts	change that branch only	original case and four direction checks pass

“The function is broken” is not a useful hypothesis. “The `LEFT` branch uses the same update as `RIGHT`” is specific, testable, and can be disproved.

3. Your route through the unit

Step	Lesson	What you will be able to do
1	Understand Failures and Read Tracebacks	Tell syntax failures, runtime exceptions, and wrong results apart; extract evidence from tracebacks and common exception types.
2	Raise Useful Errors and Handle Expected Failures	Design narrow exception boundaries, add context without erasing causes, and allow unexpected defects to remain visible.
3	Debug One Hypothesis at a Time	Reproduce a wrong result, localize its first divergence, run controlled experiments, and verify a root-cause repair.
4	Pause, Inspect, and Shrink a Bug	Choose between prints, assertions, a debugger, and a minimal reproduction; inspect program state without changing it accidentally.
5	Unit Challenge: Repair the Glitched Arcade	Repair a command-driven game while preserving useful failures and proving the final behavior with assertions.

The first two lessons concentrate on failures Python reports. The next two concentrate on investigations you must lead. The challenge combines both.

4. Bring forward the habits from Unit 7

Unit 7 separated a problem statement from its examples, acceptance rules, and algorithm. Keep that separation here. A **logic bug** is visible only when you have an oracle: a rule, example, assertion, or independently calculated result that says what should have happened.

Before editing code, write these four lines:

```text
Input:
Expected behavior:
Actual behavior:
Smallest known reproduction:
```

If one line is unknown, investigate that missing evidence first. Changing code before you know the expected behavior can produce a different program without producing a correct one.

Unit 9 will add files, paths, and external data. This unit deliberately uses small in-memory values so that an input format or operating-system path cannot distract from the failure itself.

5. Set up two useful practice environments

Use the generated notebook for quick experiments, traceback reading, and assertions. Run cells from top to bottom when confirming a repair; old notebook state can make deleted code or stale variables appear to work.

For the debugger lesson, also save a complete example as a local `.py` file. You can run it normally before introducing the debugger:

```bash
python courier.py
```

Then run the same file under Python's debugger:

```bash
python -m pdb courier.py
```

The goal is not to memorize a particular editor. It is to understand pauses, frames, variables, stepping, and continuation well enough to recognize them in Python's `pdb`, VS Code, or another debugger.

Do not debug a secret in public evidence

Tracebacks, debugger displays, screenshots, and minimal reproductions can expose passwords, tokens, personal records, and local paths. Replace sensitive values with harmless examples while preserving the data shape that triggers the bug. :::

6. Budget time for investigation, not only reading

A realistic pace is four to five hours per main lesson and one to two hours for the challenge. That includes typing examples, predicting results, deliberately triggering failures, completing labs, and recording one investigation.

For every repair clinic:

1. reproduce the exact failure without editing;
2. name the category and copy the decisive evidence;
3. shrink the input or inspect the suspicious state;
4. state one hypothesis that could be false;
5. make one controlled change;
6. rerun the original reproduction;
7. add a regression check that would fail if the bug returned.

Keep one useful failure

Do not erase every failed attempt from your notes. Preserve one failure, the hypothesis it challenged, and the evidence that changed your mind. That record shows more debugging skill than a final green check by itself. :::

7. The challenge ahead

Pixel, an arcade character, must follow movement commands and reach a portal. The supplied program looks plausible but mishandles invalid commands, one direction, and an exception boundary. Progressive assertions will let you repair one subsystem at a time without needing a teacher to reveal the answer.

Your final evidence will include:

- a parser that rejects malformed commands with useful context;
- movement rules that work in all four directions;
- a game loop that continues after anticipated command mistakes without hiding unexpected programming defects;
- assertions run from a clean state; and
- a short evidence log for one repaired defect.

There is no final project or certificate in this release. Both are planned, and we will announce them when they are ready. For now, the strongest proof of progress is a program you can explain, break deliberately, repair, and rerun.

Understand Failures and Read Tracebacks

Online lesson and runnable notebook

Distinguish syntax failures, runtime exceptions, and wrong results, then use tracebacks and small inspections to find the next useful clue.

- **Level:** Python Foundations
- **Estimated time:** 4–5 hours
- **You will learn:** Classify failures, read traceback frames from the final line upward, inspect suspicious state, and distinguish a reported location from an earlier cause.
- **Practice in:** Google Colab, JupyterLab, or a local editor

A community festival uses a small program to total game scores. During setup, three versions arrive at the repair desk:

1. one never starts because a colon is missing;
2. one starts but stops while adding a score stored as text; and
3. one prints a total that is too large but reports no error.

The first useful question is not “Which line should I change?” It is “What kind of failure do I have, and what evidence comes with it?” Keep these investigation questions beside your notebook:

- Did Python begin running the program?
- If execution stopped, what are the exception type and message?
- Which call-stack frames belong to code I control?
- Which values and types were present at the deepest useful frame?
- If no exception occurred, what independent rule proves the result is wrong?

1. Three failures leave three kinds of evidence

Python source passes through parsing before its statements can run. After it starts, an operation may raise an exception. Even when every operation succeeds, the algorithm may still produce the wrong answer.

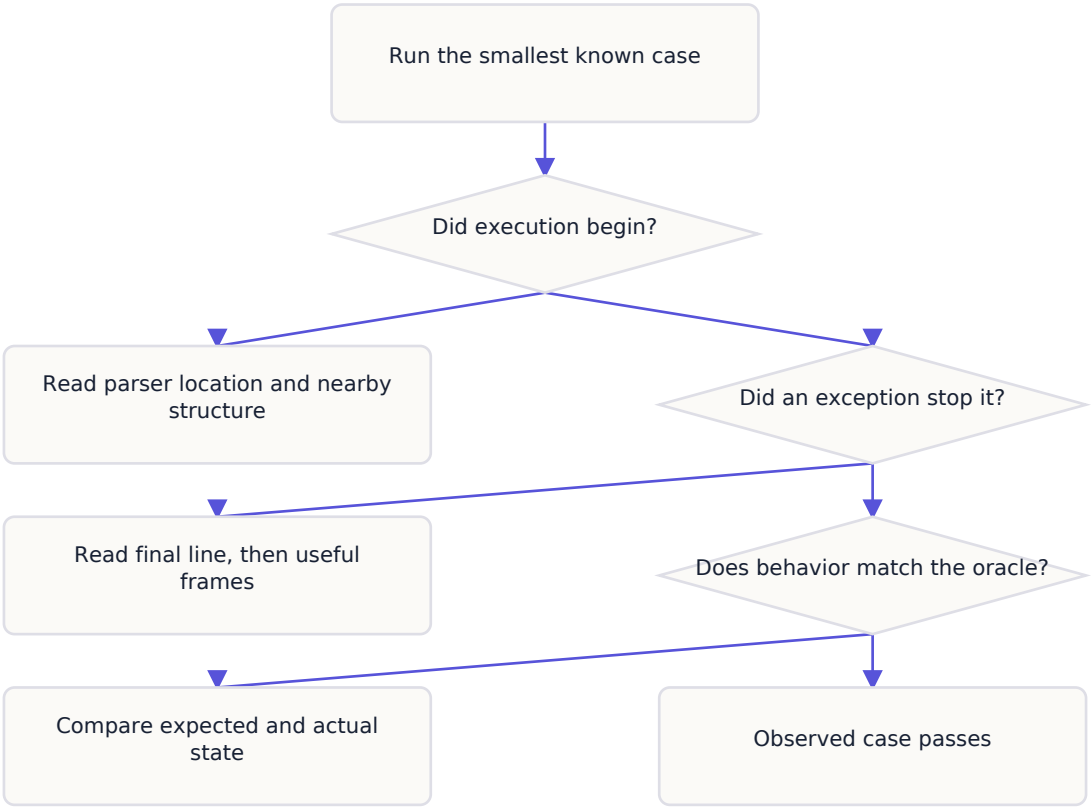
Category	What happens	Typical evidence	First useful move
syntax failure	Python cannot parse the instructions	`SyntaxError` or `IndentationError`, a line, and sometimes a caret	inspect nearby punctuation and block structure
runtime exception	execution starts, then an operation cannot continue	traceback, exception type, and message	read the final line, then inspect frames and values
logic bug	execution finishes, but behavior violates a rule	wrong output, failed assertion, or user report	state expected versus actual and shrink the case

Here is a correct baseline:

```python
scores = [8, 10, 7]
festival_total = sum(scores)
print(festival_total)
```

Python creates a list of three integers, passes it to `sum`, stores `25`, and prints `25`. That normal path gives you something to compare with each failure.

The failure category determines which evidence is available before you edit.



Did execution begin? → Read parser location and nearby structure: no

Did execution begin? → Did an exception stop it?: yes

Did an exception stop it? → Read final line, then useful frames: yes

Did an exception stop it? → Does behavior match the oracle?: no

Does behavior match the oracle? → Compare expected and actual state: no

Does behavior match the oracle? → Observed case passes: yes

An oracle is any independent statement of expected behavior: an acceptance example, calculation, assertion, specification, or trusted comparison.

2. Syntax failures happen before ordinary execution

This intended program should announce a bonus for a score of at least ten. The missing colon prevents Python from forming the `if` statement:

```python
score = 10
if score >= 10
    print("Bonus round!")
```

A local run may report:

```text
  File "festival.py", line 2
    if score >= 10
                 ^
SyntaxError: expected ':'
```

Read it in three passes:

1. `SyntaxError` names the category.
2. `expected ':'` describes what the parser needed.
3. The caret points near where parsing became impossible.

The caret is a clue, not a guarantee that the root cause is exactly underneath it. An unclosed parenthesis on an earlier line can make a later, innocent line look suspicious. Inspect the reported line **and a few lines above it**.

Indentation participates in Python's grammar. The next program has a colon, but the intended body is not indented:

```python
players = ["Ada", "Lin"]
for player in players:
print(player)
```

Python reports an `IndentationError`, which is a specialized syntax failure. The repair is to make block membership visible:

```python
players = ["Ada", "Lin"]
for player in players:
    print(player)
```

Do not wrap syntax failures in `try`/`except`. The file must be parsed before normal exception handling can run. Repair the source structure first.

Ask what Python was trying to finish

For a parser failure, scan for a missing colon, quote, closing bracket, comma, or indented body. Then run again. One syntax repair can reveal the next parser failure that was previously unreachable. :::

3. Runtime exceptions stop one path through valid code

This program is valid Python, so execution begins:

```python
def total_scores(values):
    """Return the sum of numeric scores."""
    return sum(values)
```

PYTHON (CONTINUED)

```
scores = [8, "10", 7]
print(total_scores(scores))
```

The list mixes integers with the string `"10"`. A terminal traceback resembles:

TEXT

```
Traceback (most recent call last):
  File "festival.py", line 7, in <module>
    print(total_scores(scores))
          ^^^^^^^^^^^^^^^^^^^^
  File "festival.py", line 3, in total_scores
    return sum(values)
           ^^^^^^^^^^^
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Start at the bottom, not at the first red line:

- **type:** `TypeError` says an operation received incompatible types;
- **message:** `int` and `str` were combined with addition;
- **deepest frame in our code:** `return sum(values)` is where Python noticed;
- **caller frame:** `total_scores(scores)` shows how execution arrived there.

The failing line is not necessarily the line that introduced the defect. `sum` correctly rejects the mixed list. The suspicious data was created earlier when `"10"` entered `scores`. Tracebacks tell you where execution could not continue; your investigation follows data backward to where the broken assumption began.

Checkpoint: classify before repairing

Choose the first useful evidence

Classify each situation before choosing a repair.

1. A missing colon prevents a file from starting. Which category and first move fit?

- **A.** Logic bug; add a print at the end
- **B.** Syntax failure; inspect the reported line and nearby structure
- **C.** Runtime exception; catch Exception
- **D.** Normal result; change the expected value

2. The program finishes and prints 31, while an independent calculation says 25. What evidence is decisive?

- **A.** A SyntaxError
- **B.** The color of the terminal output
- **C.** A broader except block
- **D.** The expected-versus-actual mismatch

3. Where should you begin reading a normal runtime traceback?

- **A.** At the final line containing the exception type and message

- **B.** At whichever filename is longest
- **C.** At the first library frame only
- **D.** By editing every frame at once

Write your choices before checking the answer key at the end of this book.

4. Traceback frames are a route through function calls

More than one frame is useful when functions call other functions:

```python
def apply_bonus(score, multiplier):
    """Return a score after applying a numeric multiplier."""
    return score * multiplier


def score_player(player):
    """Return one player's adjusted score."""
    return apply_bonus(player["score"], player["bonus"])


def build_leaderboard(players):
    """Return adjusted scores in input order."""
    return [score_player(player) for player in players]


players = [
    {"name": "Ada", "score": 8, "bonus": 2},
    {"name": "Lin", "score": 7, "bonus": None},
]
build_leaderboard(players)
```

The second player eventually triggers:

```text
Traceback (most recent call last):
  File "leaderboard.py", line 20, in <module>
    build_leaderboard(players)
  File "leaderboard.py", line 13, in build_leaderboard
    return [score_player(player) for player in players]
  File "leaderboard.py", line 8, in score_player
    return apply_bonus(player["score"], player["bonus"])
  File "leaderboard.py", line 3, in apply_bonus
    return score * multiplier
TypeError: unsupported operand type(s) for *: 'int' and 'NoneType'
```

Each frame is a paused layer of work:

1 module code called `build_leaderboard`;

2 the comprehension called `score_player`;

3 `score_player` supplied values to `apply_bonus`;

4 multiplication discovered `7 * None` could not proceed.

The deepest frame is often the most immediate failure. Move upward to learn who supplied its arguments and which application-level rule was violated. Skip deep library internals at first unless your own frames do not explain the inputs.

5. Inspect values without changing the case

Before repairing, make the suspicious values visible. `repr` is useful because quotes, escape characters, empty strings, and spaces remain visible:

```python
suspicious_score = " 10 "
print(repr(suspicious_score))
print(type(suspicious_score).__name__)
print(len(suspicious_score))
```

Expected output:

```text
' 10 '
str
4
```

For mappings, inspect available keys rather than assuming one exists:

```python
player = {"name": "Ada", "points": 8}
print(sorted(player.keys()))
print(player.get("score", "<missing>"))
```

For a collection, inspect shape and representative values:

```python
scores = [8, "10", 7]
print("length:", len(scores))
print("values:", repr(scores))
print("types:", [type(score).__name__ for score in scores])
```

Temporary observations should answer a question. Printing twenty unrelated variables creates noise. A targeted observation such as “What are the type and representation of the score passed to `sum`?” can confirm or disprove a hypothesis.

Preserve the reproduction while observing it

Do not call `.strip()`, replace a missing value, or convert a type merely to print it. That changes the case. First display the state as it exists; repair it only after the evidence supports a cause. :::

6. Exception names narrow the search

An exception name does not automatically tell you the repair, but it suggests which relationship to inspect.

`NameError`: a name is unavailable

```python
player_name = "Ada"

try:
    print(playre_name)
except NameError as error:
    print(type(error).__name__, "-", error)
```

Check spelling and scope. Do not create a second variable merely to match a typo; determine which name the program intended.

`TypeError`: an operation and its values do not fit

```python
score = "8"

try:
    print(score + 2)
except TypeError as error:
    print(type(score).__name__)
    print(error)
```

Inspect both types and the domain rule. The repair might be conversion at an input boundary, or it might be rejecting text instead of silently converting it.

`ValueError`: the type is acceptable, the value is not

```python
raw_score = "many"

try:
    score = int(raw_score)
except ValueError as error:
    print(repr(raw_score), "cannot become an integer")
    print(error)
```

The string type is a supported input for `int`, but these characters do not represent an integer.

`IndexError` and `KeyError`: a requested position or key is absent

```python
players = ["Ada", "Lin"]
record = {"name": "Ada", "points": 8}

try:
    print(players[2])
except IndexError as error:
    print("length:", len(players), "requested index: 2")
    print(error)

try:
    print(record["score"])
except KeyError as error:
    print("available keys:", sorted(record))
    print(error)
```

Ask whether absence is a normal case. If it is, design that behavior explicitly. If the key is required by the contract, a default may hide corrupt data.

`AttributeError`: the object lacks the requested attribute

```python
name = "Ada"

try:
    print(name.apend("!"))
except AttributeError as error:
    print(type(name).__name__)
    print(error)
```

Inspect the object type and spelling. Methods such as `append` belong to lists, and this example also misspells the intended method.

`ZeroDivisionError`: the divisor reached zero

```python
total_points = 0
player_count = 0

try:
    print(total_points / player_count)
except ZeroDivisionError as error:
    print("player_count:", player_count)
    print(error)
```

The useful question is how `player_count` became zero and what an empty tournament should mean—not how to suppress division itself.

Checkpoint: let the exception focus your inspection

Use exception names as clues

Choose the inspection that best matches the evidence.

1. A required `record["score"]` lookup raises `KeyError`. What should you inspect first?

- **A.** Whether the terminal supports colors
- **B.** The record's available keys and the required-data contract
- **C.** Only the length of an unrelated list
- **D.** How to catch every exception

2. `int("many")` raises which exception and why?

- **A.** TypeError, because strings are never accepted
- **B.** IndexError, because the string is too short
- **C.** NameError, because `int` is unavailable
- **D.** ValueError, because the string type is accepted but its content is not an integer

3. A traceback points at `sum(values)`, but `values` contains one unexpected string created earlier. Which statement is accurate?

- **A.** The reported line is where Python noticed; the defective value may have originated earlier
- **B.** `sum` must be rewritten
- **C.** The traceback proves the list was created correctly
- **D.** Every frame should be edited simultaneously

Write your choices before checking the answer key at the end of this book.

7. Chained tracebacks preserve two levels of truth

Programs sometimes add domain context to a lower-level exception. The following function explains which festival field failed while preserving `int`'s original reason:

```python
def parse_score(raw_score, player_name):
    """Return an integer score or raise a contextual ValueError."""
    try:
        return int(raw_score)
    except ValueError as error:
        raise ValueError(
            f"invalid score for player {player_name!r}: {raw_score!r}"
        ) from error


try:
    parse_score("many", "Ada")
except ValueError as error:
    print(error)
    print("original cause:", error.__cause__)
```

The resulting traceback contains the original conversion failure and the new festival-specific failure, connected by “the above exception was the direct cause.” Do not stop at the first final line when a traceback is chained. Read each exception and the relationship between them. Lesson 2 develops this design in depth.

8. Logic bugs need an oracle because Python sees valid operations

This bonus function runs successfully but awards the bonus twice:

```python
def final_score(base_score, bonus):
    """Return the base score plus one bonus."""
    subtotal = base_score + bonus
    return subtotal + bonus


actual = final_score(8, 2)
expected = 10
print("expected:", expected)
print("actual:", actual)
assert actual == expected
```

All values and operations are valid. Python has no way to infer that “one bonus” is the festival rule. The assertion turns a silent wrong result into precise evidence.

Trace the state:

Step	base_score	bonus	subtotal	returned
expected rule	8	2	10	10
actual code	8	2	10	12

The first divergence is the return expression, not the input and not the subtotal. That is a stronger localization than “something in `final_score` is wrong.” Lesson 3 turns first-divergence searches into a repeatable method.

9. Notebook state can create a fourth kind of confusion

Notebook cells can be run out of order. If you change a function cell but do not rerun it, later cells still use the old function object. If you delete a variable assignment from a cell, the old variable can remain in memory.

Use this verification routine after a repair:

1. save the notebook;
2. restart the runtime or kernel;
3. run all cells from top to bottom;
4. confirm the original failure and nearby assertions;
5. record the result only after that clean run.

A clean rerun separates program behavior from accidental session history.

10. Repair the festival scoreboard tickets

Create an evidence table with one row per ticket. For each, record the category, decisive clue, one inspection, root cause, repair, and regression check.

Ticket 1: parser failure

```python
def announce_winner(name)
    print(f"Winner: {name}")
```

Run or read the reported syntax failure, repair it, and explain why no function call could run before the repair.

Ticket 2: indentation failure

```python
for score in [8, 10, 7]:
print(score)
```

Repair the block and predict the three output lines before running.

Ticket 3: incompatible score type

```python
def add_scores(left, right):
    return left + right


try:
    add_scores(8, "10")
except TypeError as error:
    print(error)
```

Inspect `repr` and `type` for both arguments. Decide where conversion belongs if the string came from user input.

Ticket 4: unconvertible score

```python
raw_scores = ["8", "many", "7"]

for position, raw_score in enumerate(raw_scores):
    try:
        print(int(raw_score))
    except ValueError as error:
        print("position:", position, "value:", repr(raw_score))
        print(error)
```

Identify the exact element and preserve its raw representation.

Ticket 5: absent player

```python
players = ["Ada", "Lin"]

try:
    print(players[2])
except IndexError as error:
    print("length:", len(players))
    print(error)
```

Explain why the maximum valid index is one when the length is two.

Ticket 6: changed record shape

```python
record = {"player": "Ada", "points": 8}

try:
    print(record["score"])
except KeyError as error:
    print("keys:", sorted(record))
    print(error)
```

Do not add a default until you decide whether `score` is optional or the producer changed its field name.

Ticket 7: empty average

```python
def average_score(scores):
    return sum(scores) / len(scores)


try:
    average_score([])
except ZeroDivisionError as error:
    print("empty input reached average_score")
    print(error)
```

Write the intended empty-input behavior before changing the function.

Ticket 8: silent bonus defect

```python
def award_bonus(score, won_round):
    if won_round:
        score = score + 2
    return score + 2


assert award_bonus(8, False) == 8
assert award_bonus(8, True) == 10
```

Use the two cases to identify the first rule violation. Repair the cause and keep both assertions as regression checks.

Compare a repair path after investigating all eight tickets

Tickets 1 and 2 need a colon and an indented block before execution can begin. Tickets 3–7 need a contract decision as well as a technical fix: convert scores at the input boundary, decide whether absent positions/keys are expected, and define what an empty average means. For example, an explicit empty-input policy can be:

```python
def average_score(scores):
    """Return the mean score; reject an empty score collection."""
    if not scores:
        raise ValueError("cannot average an empty score collection")
    return sum(scores) / len(scores)
```

The double-bonus repair removes the unconditional extra addition:

```python
def award_bonus(score, won_round):
    if won_round:
        score = score + 2
    return score


assert award_bonus(8, False) == 8
assert award_bonus(8, True) == 10
```

Compare the evidence that led you there. A matching final function without a category, first clue, and root-cause explanation is an incomplete repair record.

Checkpoint: choose the next investigation move

Turn a failure into an investigation

Choose the move that adds evidence without hiding the problem.

1. Why does the double-bonus function need an assertion or other oracle?

- **A.** Because Python cannot execute addition
- **B.** Because the program needs more syntax errors
- **C.** Because valid operations can still violate a business rule that Python does not know
- **D.** Because every function must raise an exception

2. Which observation best investigates a suspected whitespace-and-type problem in `raw_score`?

- **A.** Print `repr(raw_score)` and `type(raw_score).__name__`
- **B.** Print every global variable ten times
- **C.** Immediately replace `raw_score` with zero
- **D.** Catch all exceptions and continue

3. What is the strongest confirmation that a notebook repair does not depend on stale state?

- **A.** One cell happened to pass before the edit
- **B.** The traceback color changed
- **C.** The notebook contains many variables
- **D.** Restart the runtime and run all cells in order, including regression checks

Write your choices before checking the answer key at the end of this book.

11. Key points for exception design

- Syntax failures happen while parsing; inspect structure before ordinary execution.
- For a runtime traceback, read the final exception type and message, then move through frames from the deepest useful application frame toward its callers.
- The reported line shows where Python noticed a broken assumption, not always where the bad value originated.
- `repr`, `type`, lengths, keys, arguments, and small state traces turn guesses into evidence.
- Exception names narrow the relationship to inspect; they do not choose the product rule for you.
- A logic bug requires an oracle and an expected-versus-actual comparison.
- A clean notebook restart is part of verification, not optional housekeeping.

References and next steps

- Python tutorial: Errors and Exceptions
- Python language reference: Exceptions
- Python built-in exception hierarchy

Next, you will decide when your own functions should raise an exception, where an application can handle it responsibly, and how to preserve the original cause while adding useful context.

Raise Useful Errors and Handle Expected Failures

Online lesson and runnable notebook

Raise precise built-in exceptions, catch only anticipated failures at informed boundaries, and preserve causes while adding useful context.

- **Level:** Python Foundations
- **Estimated time:** 4–5 hours
- **You will learn:** Raise meaningful built-in exceptions, place narrow `try` blocks, handle only failures a boundary can resolve, and chain contextual errors to their original causes.
- **Practice in:** Google Colab, JupyterLab, or a local editor

A text adventure receives commands such as `"OPEN chest"` and `"TAKE key"`. Malformed input is expected: a player may omit the target or type an unsupported action. A programming defect is not expected: if movement code uses an undefined name, the game should not pretend the player merely mistyped a command.

Good exception design preserves that difference. It answers four questions:

1. What does this function promise to return?
2. Which inputs violate that promise, and which exception explains why?
3. Where does the program know enough to recover, retry, report, or continue?
4. Which failures must remain visible because this layer cannot handle them?

1. Return normal results and raise exceptional failures

Start with a command parser whose normal result is one normalized tuple:

PYTHON

```
def parse_quest_command(text):
    """Return a normalized (action, target) command."""
    parts = text.split()
    if len(parts) != 2:
        raise ValueError("command must contain an action and a target")

    action, target = parts
    return action.upper(), target.lower()


print(parse_quest_command("OPEN Chest"))
```

Line by line:

1. `split()` turns whitespace-separated input into parts.
2. The function checks the supported shape before unpacking.
3. `raise ValueError(...)` stops this function call when the value violates its contract, even though the input is still a string.

4 The normal path returns normalized data with one predictable shape.

Do not use a magic return such as `None`, `False`, or an empty tuple unless that value is a normal, documented result. A caller can accidentally pass a magic value deeper into the program and lose the original explanation.

PYTHON
```
def find_key(inventory):
    """Return the key name if present, otherwise None."""
    for item in inventory:
        if item.endswith(" key"):
            return item
    return None
```

Here, not finding a key is an ordinary search result, so `None` is reasonable. In contrast, a command with three words violates the parser's accepted shape, so `ValueError` makes the failed contract explicit.

2. Choose an exception that describes the broken contract

For foundation-level interfaces, built-in exceptions cover many useful cases:

Exception	Use it when	Example message
`TypeError`	the object type is unsupported	`command must be a string`
`ValueError`	the type is acceptable but its value or shape is unsupported	`steps must be positive`
`KeyError`	a required mapping key is absent and mapping-style lookup is the contract	Python's normal missing-key message
`IndexError`	a requested sequence position is outside the supported range	Python's normal invalid-index message
`RuntimeError`	the operation cannot proceed because of runtime state and no more specific exception fits	`quest has not started`

Validate type and value separately when the distinction helps the caller:

PYTHON
```
def parse_repeat_count(raw_count):
    """Return a positive repeat count from an integer string."""
    if not isinstance(raw_count, str):
        raise TypeError("repeat count must be text")

    count = int(raw_count)
    if count <= 0:
        raise ValueError("repeat count must be positive")
    return count
```

`parse_repeat_count(3)` violates the type contract. `parse_repeat_count("0")` has the accepted type but violates the range rule. `parse_repeat_count("many")` lets `int` raise its own `ValueError`; a later section adds command context.

Messages should help the reader act without leaking secrets. Mention the field, rule, or position; avoid dumping credentials or an entire private record.

```python
def require_energy(energy):
    """Return energy if it can power one move."""
    if energy < 1:
        raise ValueError(f"energy must be at least 1; received {energy}")
    return energy
```

Checkpoint: design the function contract

Separate normal results from contract failures

Choose the interface behavior that communicates the rule most clearly.

1. A search may normally find no matching item. Which design can be appropriate when documented?

- **A.** Raise `SyntaxError` from valid source
- **B.** Catch every possible exception
- **C.** Return several unrelated shapes
- **D.** Return `None` for no match

2. A function accepts integers, but receives `0` where the contract requires a positive number. Which built-in exception fits?

- **A.** NameError
- **B.** IndentationError
- **C.** ImportError
- **D.** ValueError

3. Which message best helps repair an invalid command without exposing unrelated data?

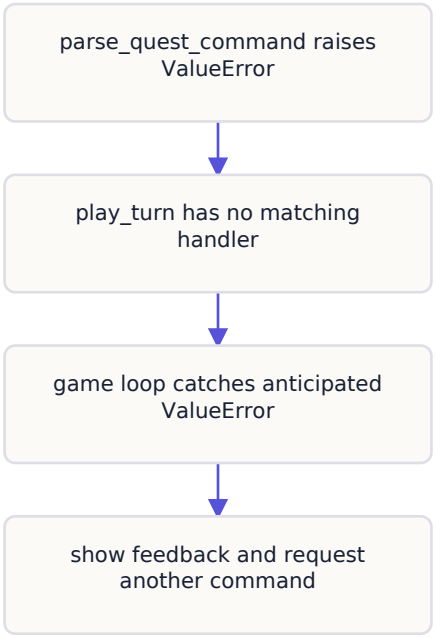
- **A.** bad
- **B.** something failed somewhere
- **C.** command at turn 4 must contain an action and a target
- **D.** all user records and tokens: ...

Write your choices before checking the answer key at the end of this book.

3. Exceptions travel upward until a caller handles them

When a function raises, its normal path stops. Python leaves that frame and returns control to its caller only if a matching `except` block exists there. Otherwise, unwinding continues up the call stack.

A low-level parser can report the violation while an outer interaction boundary decides what the player should see.



Unexpected exceptions continue past this boundary and remain visible to the developer.

Use a small example to observe propagation:

```python
def decode_steps(raw_steps):
    return int(raw_steps)


def build_move(raw_steps):
    steps = decode_steps(raw_steps)
    return {"action": "MOVE", "steps": steps}


try:
    build_move("many")
except ValueError as error:
    print("command rejected:", error)
```

`decode_steps` does not need to catch `ValueError` merely to raise the same thing. `build_move` cannot recover either. The outer boundary can turn an anticipated conversion failure into player feedback.

Catch an exception only where you can make a responsible decision:

- ask for another value;
- skip one independently invalid item while recording why;
- translate a technical failure into domain context;
- release a resource and re-raise; or
- end one operation cleanly.

If a layer can do none of those, let the exception propagate.

4. Keep the `try` block as narrow as the claim

This handler is too broad in two ways:

```python
def play_turn_too_broad(text, inventory):
    try:
        action, target = parse_quest_command(text)
        inventory.append(target)
        return f"completed {action} {target}"
    except Exception:
        return "invalid command"
```

The misspelled `.apend` raises `AttributeError`, a programming defect. The broad handler relabels it as player error. The program appears resilient while losing the very evidence needed to repair it.

A better boundary surrounds only the anticipated operation:

```python
def play_turn(text, inventory):
    try:
        action, target = parse_quest_command(text)
    except ValueError as error:
        return f"command rejected: {error}"

    inventory.append(target)
    return f"completed {action} {target}"
```

Now the `except` claim is precise: “invalid command shape is expected here.” If inventory code raises `AttributeError`, the defect stays visible.

A passing program can still be hiding failures

`except Exception: pass` does not prove recovery. It proves only that many failures were silenced. Handle a specific anticipated type, retain evidence, and keep the protected region narrow. :::

5. Match specific exceptions before general ones

Handlers are tested from top to bottom. Put more specific responses first:

```python
def parse_slot(raw_slot, slots):
    """Return the item stored at a text index."""
    try:
        index = int(raw_slot)
        return slots[index]
    except ValueError as error:
        return f"slot must be an integer: {error}"
    except IndexError as error:
        return f"slot is outside the inventory: {error}"
```

This example catches two anticipated failures from a deliberately small region. If the result is needed as data rather than UI text, a better design may let the exceptions propagate instead of returning mixed result shapes. Boundary design depends on who calls the function.

You can group exceptions only when the same response is truly appropriate:

```python
def read_required_pair(mapping):
    """Return two required integer fields or explain malformed input."""
```

PYTHON (CONTINUED)

```
    try:
        left = int(mapping["left"])
        right = int(mapping["right"])
    except (KeyError, ValueError) as error:
        raise ValueError("pair needs integer 'left' and 'right' fields") from error
    return left, right
```

Do not group types simply to write fewer lines. Ask whether the caller should react identically and whether the new message remains accurate for each cause.

6. Use `else` and `finally` for different jobs

An `else` suite runs only when the `try` suite finishes without a matching exception. It keeps success work outside the protected region:

PYTHON

```
def describe_repeat(raw_count):
    try:
        count = int(raw_count)
    except ValueError as error:
        return f"invalid repeat count: {error}"
    else:
        return f"The spell will repeat {count} times."
```

If string formatting in `else` had a defect, this handler would not mislabel it as an integer-conversion problem.

A `finally` suite runs whether the protected operation returns, raises, or is handled. Use it for cleanup that must happen:

PYTHON

```
def demonstrate_cleanup(raw_count):
    events = []
    try:
        events.append("start conversion")
        count = int(raw_count)
    except ValueError:
        events.append("conversion rejected")
    else:
        events.append(f"converted {count}")
    finally:
        events.append("close turn record")
    return events


print(demonstrate_cleanup("3"))
print(demonstrate_cleanup("many"))
```

Avoid `return` in `finally`. A return there can replace a normal return or suppress an active exception, erasing evidence:

PYTHON

```
def safe_cleanup_pattern(raw_count):
    try:
        result = int(raw_count)
    finally:
        print("cleanup happened")
    return result
```

Later units use context managers such as `with` for resource cleanup. The same principle applies: cleanup should not disguise the operation's outcome.

Checkpoint: place the boundary precisely

Protect only the operation you understand

Choose the boundary that preserves unexpected defects.

1. A game expects command conversion to fail, but an inventory method may contain a bug. What should the `try` block contain?

- **A.** The entire game loop
- **B.** Only the anticipated command parsing or conversion operation
- **C.** Every function in the program
- **D.** No observable operation

2. When does an `else` suite attached to `try` run?

- **A.** Only when the `try` suite finishes without an exception
- **B.** Only after a syntax failure
- **C.** Before the `try` suite
- **D.** Instead of every `finally` suite

3. Why should you generally avoid returning from `finally`?

- **A.** It makes integer conversion faster
- **B.** It creates a syntax error in every Python version
- **C.** It can replace another return or suppress an active exception
- **D.** It forces all exceptions to be chained

Write your choices before checking the answer key at the end of this book.

7. Add context and preserve the original cause

Low-level exceptions know technical details. Outer layers know domain details. Exception chaining preserves both:

PYTHON

```
def parse_damage(raw_damage, turn_number):
    """Return positive damage or raise a turn-specific ValueError."""
    try:
        damage = int(raw_damage)
    except ValueError as error:
        raise ValueError(
            f"turn {turn_number}: damage must be an integer; got {raw_damage!r}"
        ) from error

    if damage <= 0:
        raise ValueError(
            f"turn {turn_number}: damage must be positive; got {damage}"
        )
    return damage
```

There are two `ValueError` paths, but only conversion chains an underlying cause. A nonpositive integer was parsed successfully; the function itself rejects the domain value, so there is no hidden conversion exception to invent.

Inspect the chain:

PYTHON
```
try:
    parse_damage("many", 4)
except ValueError as error:
    print("context:", error)
    print("cause type:", type(error.__cause__).__name__)
    print("cause message:", error.__cause__)
```

Use a bare `raise` inside a handler when you record evidence but cannot actually recover:

PYTHON
```
def parse_with_observation(raw_damage):
    try:
        return int(raw_damage)
    except ValueError:
        print("conversion failed for", repr(raw_damage))
        raise
```

Bare `raise` preserves the current exception and traceback. `raise error` can alter traceback details; creating an unrelated new exception discards the causal link unless you chain it explicitly.

8. Assertions check developer assumptions, not public input

An assertion is excellent for an internal condition that should be true if the program is correct:

PYTHON
```
def apply_quest_rewards(rewards):
    total = sum(rewards)
    assert total >= 0, "reward total should never be negative"
    return total
```

Do not make `assert` the only validation for player input:

PYTHON
```
def set_volume(volume):
    """Return a supported game volume from 0 through 10."""
    if not isinstance(volume, int):
        raise TypeError("volume must be an integer")
    if not 0 <= volume <= 10:
        raise ValueError("volume must be between 0 and 10")
    return volume
```

Assertions can be disabled with Python optimization settings, and `AssertionError` does not communicate the public input category as precisely. Use explicit exceptions for an interface contract; use assertions to make developer assumptions executable.

9. EAFP and LBYL are choices, not slogans

Python code often uses **EAFP**: attempt the operation and handle an anticipated exception (“easier to ask forgiveness than permission”). **LBYL** checks a condition first (“look before you leap”). Neither style means catch everything.

LBYL can clearly express a domain rule:

```python
def spend_energy(energy, cost):
    if cost < 0:
        raise ValueError("cost cannot be negative")
    if cost > energy:
        raise ValueError("not enough energy")
    return energy - cost
```

EAFP can avoid duplicating an operation's own validation:

```python
def convert_coordinates(raw_x, raw_y):
    try:
        return int(raw_x), int(raw_y)
    except ValueError as error:
        raise ValueError("coordinates must be integers") from error
```

Choose based on clarity, supported races or state changes, and the contract. Do not probe with an unreliable check and then assume the later operation cannot fail. Do not catch a broad type merely to call the code "Pythonic."

Custom exception classes become valuable when callers need to distinguish domain failures that built-in types cannot express cleanly. Unit 11 introduces classes; this unit deliberately practices precise built-in exceptions first.

10. Build a reliable quest command boundary

Use this input set:

```python
quest_commands = [
    "OPEN chest",
    "TAKE",
    "MOVE north",
    "CAST lantern",
    "TAKE key",
]
```

Implement the contract in stages:

```python
SUPPORTED_ACTIONS = {"OPEN", "TAKE", "MOVE"}


def parse_quest_command(text, turn_number):
    """Return normalized (action, target) or raise contextual ValueError."""
    if not isinstance(text, str):
        raise TypeError(f"turn {turn_number}: command must be text")

    parts = text.split()
    if len(parts) != 2:
        raise ValueError(
            f"turn {turn_number}: command needs exactly two words"
        )

    action, target = parts
    action = action.upper()
    target = target.lower()
    if action not in SUPPORTED_ACTIONS:
```

PYTHON (CONTINUED)

```
        raise ValueError(
            f"turn {turn_number}: unsupported action {action!r}"
        )
    return action, target
```

Check the parser independently:

PYTHON

```
assert parse_quest_command("open Chest", 1) == ("OPEN", "chest")
assert parse_quest_command("TAKE key", 2) == ("TAKE", "key")

try:
    parse_quest_command("TAKE", 3)
except ValueError as error:
    assert "turn 3" in str(error)
    assert "two words" in str(error)
else:
    raise AssertionError("one-word command should be rejected")
```

Then add the boundary that can continue after anticipated command mistakes:

PYTHON

```
def collect_quest_commands(commands):
    """Return accepted commands and player-readable rejection messages."""
    accepted = []
    rejected = []

    for turn_number, text in enumerate(commands, start=1):
        try:
            command = parse_quest_command(text, turn_number)
        except (TypeError, ValueError) as error:
            rejected.append(str(error))
        else:
            accepted.append(command)

    return accepted, rejected


accepted, rejected = collect_quest_commands(quest_commands)
assert accepted == [
    ("OPEN", "chest"),
    ("MOVE", "north"),
    ("TAKE", "key"),
]
assert len(rejected) == 2
assert all("turn" in message for message in rejected)
```

Try these controlled changes:

1. replace one string with `None`; confirm `TypeError` is reported with its turn;
2. add extra spaces around a valid command; explain why `split()` accepts it;
3. introduce an `AttributeError` after parsing and verify this boundary does not hide it;
4. change the supported-action set and rerun all checks.

Show one way to verify the unexpected-defect boundary

Use a temporary collaborator that fails **after** parsing, then confirm the failure is not converted into a command rejection:

```python
def apply_parsed_command(command):
    """Represent later game behavior with a deliberate unexpected defect."""
    action, target = command
    raise AttributeError(f"unfinished handler for {action} {target}")


def run_one_turn(text, turn_number):
    try:
        command = parse_quest_command(text, turn_number)
    except (TypeError, ValueError) as error:
        return f"command rejected: {error}"
    return apply_parsed_command(command)


try:
    run_one_turn("OPEN chest", 1)
except AttributeError as error:
    print("unexpected defect remained visible:", error)
```

The parsing handler is still narrow. Remove the deliberate defect after the experiment; it is evidence about the boundary, not finished game behavior.

Checkpoint: preserve the useful failure

Keep context without losing the cause

Choose the design that makes failures actionable and honest.

1. Why use `raise ValueError("turn 4: ...") from error` after `int` fails?

- **A.** To hide the conversion failure
- **B.** To add turn context while preserving the conversion error as the cause
- **C.** To make the value valid
- **D.** To continue from the failed line automatically

2. Why use explicit `TypeError` or `ValueError` rather than only `assert` for public input?

- **A.** Assertions always convert text
- **B.** Assertions catch every exception
- **C.** Explicit validation remains part of the interface and identifies the contract category
- **D.** Explicit validation prevents all programming bugs

3. The command parser succeeds, then inventory code raises an unexpected `AttributeError`. What should the command boundary do?

- **A.** Relabel it as invalid player input
- **B.** Silence it and report success
- **C.** Return an empty command
- **D.** Let it propagate so the programming defect remains visible

Write your choices before checking the answer key at the end of this book.

11. Key points for debugging

- Return ordinary outcomes; raise when a function cannot honor its documented contract.
- Choose a specific built-in exception and a safe message that names the broken rule or location.
- Let exceptions propagate until a boundary knows how to recover, report, retry, skip, clean up, or terminate responsibly.
- Keep `try` regions narrow and handlers specific so unrelated defects remain visible.
- Use `else` for success work and `finally` for unavoidable cleanup; do not erase results or exceptions with a `finally` return.
- Chain contextual exceptions with `raise ... from error`, and use bare `raise` when recording evidence before re-raising the current exception.
- Use explicit validation for public input and assertions for developer assumptions.

References and next steps

- Python tutorial: Handling Exceptions
- Python tutorial: Raising Exceptions
- Python language reference: The `try` statement
- Python built-in exceptions

The next lesson begins with a program that raises no exception at all. You will use expected-versus-actual evidence, state traces, minimal cases, and controlled experiments to locate the first wrong decision.

Debug One Hypothesis at a Time

Online lesson and runnable notebook

Reproduce wrong behavior, localize its first divergence, test one falsifiable hypothesis, repair the cause, and verify against regressions.

- **Level:** Python Foundations
- **Estimated time:** 4–5 hours
- **You will learn:** Turn a wrong result into a stable reproduction, localize its first divergence, test falsifiable hypotheses, and verify a root-cause repair with regression evidence.
- **Practice in:** Google Colab, JupyterLab, or a local editor

A weekend tournament awards points for wins and a streak bonus for consecutive wins. The following function produces a plausible total, raises no exception, and is wrong:

```python
def tournament_score(results):
    """Return 3 points per win plus 2 for each win after the first in a streak."""
    total = 0
    streak = 0

    for result in results:
        if result == "W":
            streak += 1
            total += 3 + (2 * streak)
        else:
            streak = 0

    return total


print(tournament_score(["W", "W", "L", "W"]))
```

The correct total is `11`: three wins contribute nine points, and only the second consecutive win contributes a two-point streak bonus. The function returns `17`. This lesson is about the work between “wrong” and “fixed.”

Keep one rule throughout: **each experiment should distinguish between possible causes**. Editing several lines and seeing a pass tells you less than changing one cause and predicting exactly what new evidence should appear.

1. Freeze the failure before touching the code

A reproducible bug has a known starting state, action, expected result, and actual result. Turn the report into an assertion:

```python
failing_results = ["W", "W", "L", "W"]
expected_score = 11
actual_score = tournament_score(failing_results)

print("input:", failing_results)
print("expected:", expected_score)
print("actual:", actual_score)
assert actual_score == expected_score
```

Do not repair yet. Record:

```text
Reproduction: tournament_score(["W", "W", "L", "W"])
Expected: 11, calculated from 3 + (3 + 2) + 0 + 3
Actual: 17
Environment: clean notebook run, current saved function
Frequency: every clean run
```

If a report cannot be reproduced, that is evidence—not permission to guess. Check the exact input representation, random seed, call order, Python version, saved file, working directory, configuration, and notebook state. Change one environmental factor at a time.

2. Establish a control case and nearby boundaries

A **control** is a simple case expected to work. It shows which parts of the program are not yet implicated:

```python
assert tournament_score([]) == 0
assert tournament_score(["L"]) == 0
```

Now try the smallest win case:

```python
print(tournament_score(["W"]))
```

The contract says one win earns `3`, but the function returns `5`. That smaller failure removes loss/reset behavior from suspicion. The defect is already present on the first win.

Build a behavior table before editing:

Input	Expected	Actual before repair	Rule isolated
`[]`	0	0	empty tournament
`['L']`	0	0	a loss scores nothing
`['W']`	3	5	first win has no streak bonus
`['W', 'W']`	8	12	second win adds one bonus
`['W', 'L', 'W']`	6	10	loss resets the streak

The one-win case is a minimal reproduction: removing its only event makes the failure disappear. Smaller input reduces the number of states you must explain.

A smaller failure is a sharper question

Minimization is not only for bug reports. It is an investigation tool. Remove one item, branch, function call, or configuration detail and ask whether the same rule still fails. :::

3. Separate facts from hypotheses

Facts come directly from observations or the contract:

- `tournament_score(["W"])` returns `5` on a clean run;
- the rule awards `3` for a first win;
- no loss branch runs for that input;
- `streak` begins at zero.

Hypotheses are proposed explanations:

- **H1:** `streak` is not initialized to zero.
- **H2:** the bonus formula applies to the first win instead of only later wins.
- **H3:** the loss branch incorrectly adds points.

H1 and H3 conflict with visible evidence: the source initializes zero, and the minimal case never enters the loss branch. H2 predicts exactly the observed two extra points.

A useful hypothesis names a mechanism and a predicted observation:

If the formula uses the new streak length directly, then the first win will add `2 * 1`; printing the base and bonus separately will show base `3`, bonus `2` before total becomes `5`.

“The loop is weird” cannot be meaningfully disproved. “The first win receives a bonus because the calculation uses `streak` rather than `streak - 1`” can.

Checkpoint: make the failure testable

Turn a report into stable evidence

Choose the move that makes the investigation smaller and more repeatable.

1. What should happen before editing a reported wrong result?

- **A.** Rewrite the whole function
- **B.** Delete every assertion
- **C.** Catch all exceptions
- **D.** Capture exact input, expected result, actual result, and a repeatable run

2. Why is `['W']` more useful than a long tournament when both expose the same bonus defect?

- **A.** It guarantees no other bugs exist
- **B.** It changes the contract
- **C.** It removes unrelated state and branches while preserving the failing rule
- **D.** It makes assertions unnecessary

3. Which statement is the strongest debugging hypothesis?

- **A.** The code is bad
- **B.** Python is probably confused
- **C.** Change things until it passes
- **D.** The first win gets a bonus because the formula uses streak 1; separate base and bonus should reveal +2

Write your choices before checking the answer key at the end of this book.

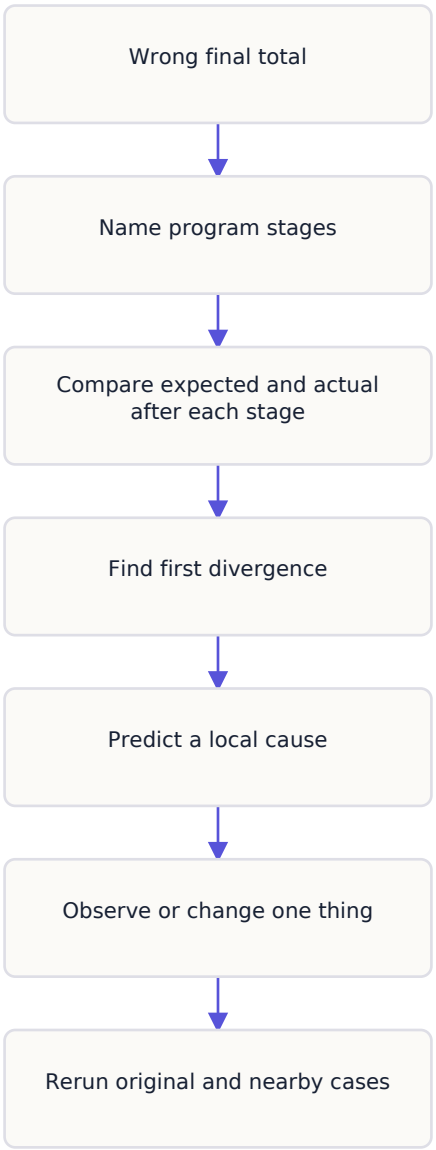
4. Localize the first wrong stage

For larger programs, identify the first stage where actual behavior differs from expected behavior. Model the scorer as a pipeline:

```text
raw results -> iterate one result -> update streak -> calculate bonus -> update total
```

The input is already a clean list, iteration sees the correct first item, and the streak becomes one as expected. The first divergence is bonus calculation. Do not edit parsing, iteration, or reset code when their observed states still match the contract.

Debugging moves from a broad failing outcome toward the first state transition that violates a known rule.



Later wrong values can be consequences. Repair the earliest supported cause, not every downstream symptom.

Function boundaries provide natural checkpoints. Suppose a leaderboard has three stages:

```python
def normalize_results(results):
    return [result.strip().upper() for result in results]


def score_results(results):
    return tournament_score(results)


def build_summary(raw_results):
    normalized = normalize_results(raw_results)
    score = score_results(normalized)
    return {"results": normalized, "score": score}
```

Check outputs at each boundary:

```python
raw = [" w ", "W", "l", "W"]
normalized = normalize_results(raw)
print("normalized:", normalized)
assert normalized == ["W", "W", "L", "W"]

score = score_results(normalized)
print("score:", score)
assert score == 11
```

If normalization passes and scoring fails, the first broken boundary lies after normalization. That does not prove every normalizer case is correct; it localizes this reproduction.

5. Trace state when the problem lives inside a loop

A state trace records selected values after each meaningful transition. Create an instrumented version without changing the scoring decisions:

```python
def trace_tournament_score(results):
    total = 0
    streak = 0
    trace = []

    for turn, result in enumerate(results, start=1):
        before_total = total
        if result == "W":
            streak += 1
            base = 3
            bonus = 2 * streak
            total += base + bonus
        else:
            streak = 0
            base = 0
            bonus = 0

        trace.append(
            {
                "turn": turn,
                "result": result,
                "streak": streak,
                "base": base,
```

PYTHON (CONTINUED)

```
                "bonus": bonus,
                "before": before_total,
                "after": total,
            }
        )

    return total, trace


actual, trace = trace_tournament_score(["W"])
print(actual)
print(trace[0])
```

The trace should show `base: 3`, `bonus: 2`, `after: 5`. That observation supports H2. It also falsifies an alternative claim that addition happens twice in separate statements.

Choose only variables needed for the hypothesis. Good instrumentation might show a loop index, current item, branch chosen, before value, delta, and after value. Printing entire nested objects on every iteration can bury the transition you need.

Remove or convert temporary prints after the repair. Preserve the valuable rule as an assertion so future runs check it automatically.

6. Change one cause and predict the result

The contract says a streak bonus applies only to wins **after the first**. If `streak` is one, bonus count should be zero; if streak is two, bonus count should be one. Repair only that formula:

PYTHON

```
def tournament_score(results):
    """Return 3 points per win plus 2 for each win after the first in a streak."""
    total = 0
    streak = 0

    for result in results:
        if result == "W":
            streak += 1
            bonus_wins = streak - 1
            total += 3 + (2 * bonus_wins)
        else:
            streak = 0

    return total
```

Before running, predict:

- `['W']` becomes `3`;
- `['W', 'W']` becomes `8`;
- the original case becomes `11`;
- loss-only controls remain zero.

Then execute the exact checks:

PYTHON

```
assert tournament_score([]) == 0
assert tournament_score(["L"]) == 0
assert tournament_score(["W"]) == 3
assert tournament_score(["W", "W"]) == 8
```

PYTHON (CONTINUED)

```
assert tournament_score(["W", "L", "W"]) == 6
assert tournament_score(["W", "W", "L", "W"]) == 11
```

A matching prediction is evidence that the causal model improved. If the original case passes but a control regresses, the change is not complete.

Checkpoint: locate before you repair

Find the first divergence

Choose the experiment that best separates causes from consequences.

1. Normalization output matches the expected list, but the score is wrong. What does that establish for this reproduction?

- **A.** Every possible normalizer input is correct
- **B.** The first observed divergence occurs after the normalization boundary
- **C.** The score must be changed manually
- **D.** No further evidence is needed

2. Which trace is most focused for a suspected loop bonus formula?

- **A.** Current result, streak, base, bonus, total before, and total after
- **B.** Every global and imported object
- **C.** Only a message saying "loop ran"
- **D.** A different random input on every run

3. Why predict the effect of one formula change before running?

- **A.** To guarantee syntax can never fail
- **B.** To avoid keeping regression checks
- **C.** To test whether the proposed cause explains the observed behavior
- **D.** To make all alternate hypotheses true

Write your choices before checking the answer key at the end of this book.

7. Repair the cause, not the visible symptom

A symptom patch could subtract six from the original result:

PYTHON

```
def patch_one_example(results):
    return tournament_score(results) - 6
```

It might match one reported case while breaking an empty tournament and every other shape. Another weak patch might special-case the exact failing list. Both encode examples rather than the streak rule.

A root-cause repair changes the earliest wrong decision supported by evidence. It should explain why all nearby cases now work:

- first win: `streak - 1` is zero;
- second consecutive win: it is one;
- a loss resets streak to zero;
- a later isolated win again receives no bonus.

After repairing, ask what other behavior shares the changed line. Test long streaks, repeated losses, empty input, and alternating results. A fix can solve the report while exposing a neighboring boundary.

8. Keep regression evidence and clean the investigation

A regression check is a small executable example that would fail if the defect returned. The strongest one is often the minimal reproduction:

PYTHON
```
def test_first_win_has_no_streak_bonus():
    assert tournament_score(["W"]) == 3


test_first_win_has_no_streak_bonus()
```

Unit 13 will organize these functions with pytest. Right now, naming the rule and running the assertion is enough to preserve learning.

Complete the repair with a clean verification:

1. remove unrelated experimental changes;
2. keep a concise assertion for the original defect;
3. run the minimal case, original case, controls, and nearby boundaries;
4. restart the notebook runtime and run top to bottom, or rerun the saved script;
5. record the cause and why the repair matches the contract.

Do not describe a repair only as “changed line 8.” Line numbers move. Record the behavioral reason: “The formula treated the first win as an additional streak win; subtracting one from the streak count makes bonuses begin at win two.”

9. Intermittent failures need controlled repetition

Suppose a game selects a random prize and a test sometimes fails. Repeatedly clicking Run creates observations but not a stable experiment. Control the random input or inject it:

PYTHON
```
def prize_message(prize):
    if prize == "coin":
        return "You found a coin"
    if prize == "map":
        return "You found a map"
    return "Mystery prize"


assert prize_message("coin") == "You found a coin"
assert prize_message("map") == "You found a map"
```

For time-dependent, network, or environment-dependent behavior, record the varying dependency and substitute a controlled value when the design permits. Unit 13 covers test doubles and dependency injection more fully.

10. Use AI suggestions as hypotheses, not evidence

An assistant can propose suspicious lines, smaller examples, or possible assertions. It cannot observe your exact runtime unless you supply accurate, safe evidence, and a plausible explanation can still be wrong.

Use this workflow:

1. remove secrets and unrelated private data;
2. provide the minimal complete reproduction, exact traceback or output, expected behavior, environment, and attempts already made;
3. ask for several falsifiable hypotheses and an experiment for each;
4. inspect every suggested change before running it;
5. run one controlled experiment locally;
6. keep the repair only when original and regression cases support it.

Never paste tokens, passwords, private student records, or proprietary data into an external tool. Replace values while preserving the shape that triggers the failure.

11. Investigate a second scoreboard defect

The tournament now supports a three-point comeback bonus after a loss followed by two wins. This implementation silently awards it too soon:

```python
def comeback_score(results):
    total = 0
    previous = None
    wins_after_loss = 0

    for result in results:
        if result == "W":
            total += 3
            if previous == "L":
                wins_after_loss += 1
            if wins_after_loss == 1:
                total += 3
        else:
            wins_after_loss = 0
        previous = result

    return total
```

The rule says `['L', 'W']` earns `3`, while `['L', 'W', 'W']` earns `9`. Conduct the investigation rather than jumping to a patch:

1. freeze both assertions and record actual results;
2. find a passing control;
3. shrink any longer reported failure;

4 list facts separately from at least three hypotheses;

5 trace `result`, `previous`, `wins_after_loss`, and `total` per turn;

6 mark the first divergence;

7 run one experiment that falsifies a hypothesis;

8 make one root-cause repair;

9 verify empty, loss-only, isolated-win, qualifying-comeback, and repeated-win cases from a clean state.

Use this evidence table:

Stage	Expected	Actual	Evidence	Next hypothesis
reproduction				
smallest case				
first divergence				
controlled experiment				
clean verification				

Compare a root-cause repair after completing the investigation

One clear model remembers whether a loss has armed a comeback and counts wins until the two-win requirement is satisfied:

PYTHON

```
def comeback_score(results):
    total = 0
    comeback_armed = False
    wins_after_loss = 0

    for result in results:
        if result == "L":
            comeback_armed = True
            wins_after_loss = 0
            continue

        total += 3
        if comeback_armed:
            wins_after_loss += 1
            if wins_after_loss == 2:
                total += 3
                comeback_armed = False
                wins_after_loss = 0

    return total


assert comeback_score([]) == 0
assert comeback_score(["W"]) == 3
assert comeback_score(["L", "W"]) == 3
assert comeback_score(["L", "W", "W"]) == 9
assert comeback_score(["L", "W", "W", "W"]) == 12
```

This is one contract-consistent design, not the only syntax that can work. Your record should still show which observation falsified a hypothesis and why the state stops awarding the comeback after it is earned.

Checkpoint: close the investigation responsibly

Prove the repair survives

Choose evidence that distinguishes a root-cause repair from a lucky pass.

1. Which case is especially valuable as a regression check?

- **A.** A new random case with no expected value
- **B.** The minimal input that exposed the original broken rule
- **C.** Only the largest available input
- **D.** A screenshot without code or output

2. What distinguishes a root-cause repair from subtracting a constant from one wrong total?

- **A.** It implements the underlying rule and explains nearby cases
- **B.** It avoids every assertion
- **C.** It changes more lines
- **D.** It catches more exceptions

3. How should an AI-generated debugging suggestion be treated?

- **A.** As proof that the bug is solved
- **B.** As permission to share secrets
- **C.** As a hypothesis to inspect and test with local evidence
- **D.** As a replacement for expected behavior

Write your choices before checking the answer key at the end of this book.

12. Key points and your reusable debugging record

For future lessons and projects, keep this compact template:

```text
Failure statement:
Environment and clean-run steps:
Smallest reproduction:
Expected result and oracle:
Actual result or exception:
Passing control:
Facts:
Candidate hypotheses:
First divergence:
Experiment and prediction:
Observed result:
Root-cause repair:
Regression cases:
Clean verification:
```

The record is not bureaucracy. It prevents repeated guesses, supports a teammate who joins later, and makes your own reasoning inspectable.

References and next steps

- Python tutorial: Errors and Exceptions
- Python `repr` built-in
- Python `assert` statement

Next, you will pause a running program, inspect stack frames and local variables, step across function calls, and reduce a realistic report into a minimal reproduction someone else can run.

Pause, Inspect, and Shrink a Bug

Online lesson and runnable notebook

Choose a debugging tool, pause before a suspicious line, inspect frames and state, and reduce a failure to a safe complete reproduction.

- **Level:** Python Foundations
- **Estimated time:** 4–5 hours
- **You will learn:** Choose between prints, assertions, traces, and a debugger; pause before a suspicious operation; inspect frames and variables; and build a minimal, complete, safe reproduction.
- **Practice in:** A generated notebook for analysis and a local `.py` file for interactive debugging

A robot courier should deliver three parcels and finish with seven energy points. Instead, it finishes with four:

PYTHON

```
def delivery_cost(distance, fragile):
    """Return energy spent for one delivery."""
    cost = distance * 2
    if fragile:
        cost += 1
    return cost


def run_route(parcels, starting_energy):
    """Return remaining energy after delivering every parcel."""
    energy = starting_energy
    for parcel in parcels:
        cost = delivery_cost(parcel["distance"], parcel["fragile"])
        energy -= cost
        energy -= 1
    return energy


parcels = [
    {"name": "map", "distance": 1, "fragile": False},
    {"name": "lamp", "distance": 2, "fragile": True},
    {"name": "key", "distance": 1, "fragile": False},
]

assert run_route(parcels, 16) == 7
```

You could add prints. You could pause execution. You could shrink the route to one parcel. Good debugging is not loyalty to one tool; it is choosing the least complicated tool that can answer the current question.

1. Match the tool to the uncertainty

Tool	Best question	Strength	Risk
traceback	where did an exception stop, and through which calls?	already produced, structured	absent for silent wrong results
focused `print`/`repr`	what value reaches this point?	fast and portable	noisy output or accidental state changes
assertion	which rule first becomes false?	executable and reusable	needs a known expected condition
state trace	how does selected state change across a loop?	exposes transitions	can collect too much data
debugger	what are locals and call frames at this exact moment?	pause, inspect, step without adding many prints	stepping without a question becomes wandering
minimal reproduction	what is essential to trigger the failure?	removes noise and aids sharing	removing too much can remove the cause

Start with the smallest question. If one assertion identifies the bad transition, a debugger is unnecessary. If values change across nested calls and a loop, pausing and stepping may be clearer than adding fifteen prints.

Write the debugger question first

Examples: “What is `cost` immediately before energy changes?” or “Which call first turns energy below the expected value?” A breakpoint without a question is only a pause. :::

2. A breakpoint pauses before the highlighted line executes

Save the opening example as `courier.py`, including a final print:

```python
print("remaining energy:", run_route(parcels, 16))
```

In VS Code, select the gutter beside this line inside `run_route`:

```python
energy -= cost
```

Start a Python debugging session. When execution reaches the breakpoint, that line has **not run yet**. Inspect:

- `parcel`: the current mapping;
- `cost`: the value returned by `delivery_cost`;
- `energy`: the value before subtraction;
- the call stack: module code called `run_route`, which is now paused.

That before/after distinction prevents off-by-one confusion. Step over the line once, then compare energy again.

The four movements through paused code

Debugger interfaces use similar actions:

- **Continue** runs until another breakpoint, exception pause, or program end.
- **Step over** executes the current line and pauses at the next line in the current frame; called functions run without pausing inside them.
- **Step into** enters a function called by the current line so you can inspect its first executable step.
- **Step out** finishes the current function and pauses back in its caller.

For the courier:

1 pause before the call to `delivery_cost`;

2 step into it when you need to verify its formula;

3 step out once the return value is understood;

4 step over each energy subtraction;

5 continue to the next parcel after recording the transition.

Stepping every library line is rarely useful. Move deliberately between application frames that can answer your hypothesis.

3. Inspect locals and the call stack together

A local variable makes sense in the context of its frame. If paused inside `delivery_cost`, locals include `distance`, `fragile`, and `cost`. The caller's frame contains `parcel` and `energy`.

Use the call-stack panel to select each frame and compare the interface:

Frame	Inputs or locals to inspect	Contract question
`delivery_cost`	`distance`, `fragile`, `cost`	does one parcel cost match the formula?
`run_route`	`parcel`, `cost`, `energy`	is the returned cost applied exactly once?
module	`parcels`, starting value	did the scenario supply the intended data?

Expressions entered in a debug console can have side effects. Prefer simple observations such as `repr(parcel)`, `cost`, `energy`, or `type(parcel).__name__`. Avoid calling methods that mutate a list or mapping unless mutation is the planned experiment.

Inspect before you edit state

Changing a variable in the debug console can make the current run pass without repairing the source. If you deliberately modify state, record it as an experiment and restart before verifying the actual fix.
:::

Checkpoint: control the pause

Know what the debugger has and has not executed

Choose the action that answers the stated question with the least wandering.

1. Execution pauses on `energy -= cost`. What does `energy` normally show at that moment?

- **A.** The value before that highlighted subtraction executes
- **B.** Always zero
- **C.** The final value after the entire loop
- **D.** A syntax error

2. You need to inspect how `delivery_cost(...)` calculates its return. Which action fits?

- **A.** Stop and delete the function
- **B.** Continue to program end
- **C.** Step into the function call
- **D.** Change every local variable

3. Why select different call-stack frames while paused?

- **A.** To rewrite Python's library
- **B.** To inspect locals in the function that owns them and compare caller/callee contracts
- **C.** To make the traceback disappear
- **D.** To skip all verification

Write your choices before checking the answer key at the end of this book.

4. Stop only on the iteration that matters

The third parcel may be the only failing case in a hundred-item route. Clicking Continue ninety-nine times is not an investigation strategy. Use a **conditional breakpoint** when your editor supports it, for example:

```text
parcel["name"] == "key"
```

The debugger evaluates that condition at the breakpoint and pauses only when it is true. Other useful conditions include:

```text
energy < 5
cost != expected_cost
turn == 7
```

Keep conditions free of side effects. A condition should observe the program, not modify it.

You can also encode a temporary condition in source:

```python
for parcel in parcels:
    cost = delivery_cost(parcel["distance"], parcel["fragile"])
    if parcel["name"] == "key":
        print("key parcel state:", repr(parcel), cost)
```

The source version is portable and easy to show in a notebook. An editor conditional breakpoint keeps production logic untouched. Choose based on the environment and whether the observation should remain in code.

5. Use `breakpoint()` and `pdb` in a local script

Python's built-in `breakpoint()` enters the configured debugger at that call site. Put it in a local script only while investigating:

```python
def run_route_with_pause(parcels, starting_energy):
    energy = starting_energy
    for parcel in parcels:
        cost = delivery_cost(parcel["distance"], parcel["fragile"])
        breakpoint()
        energy -= cost
        energy -= 1
    return energy
```

Do **not** run that cell in hosted course notebooks unless you know how the environment handles interactive debugging. Save a complete script and run it locally instead.

You can also start any saved script under Python's standard debugger without editing the file:

```bash
python -m pdb courier.py
```

At the `(Pdb)` prompt, these commands cover a focused first session:

Command	Meaning
`l`	list source around the current line
`p expression`	print one expression
`pp expression`	pretty-print one expression
`n`	next line in the current frame (step over)
`s`	step into a called function
`r`	run until the current function returns (step out)
`c`	continue until another stop
`where`	show the current stack
`up` / `down`	select an older/newer frame
`q`	quit the debugging session

A short session might look like:

```text
(Pdb) p parcel
{'name': 'map', 'distance': 1, 'fragile': False}
(Pdb) p cost
2
```

TEXT (CONTINUED)

```
(Pdb) p energy
16
(Pdb) n
(Pdb) p energy
14
```

The command letters are less important than the reasoning: pause before the transition, inspect inputs, execute one operation, inspect the after-state, and compare it with the expected transition.

Remove stray `breakpoint()` calls after investigating. A forgotten breakpoint can halt another user's run unexpectedly.

6. The courier's first divergence

Trace the opening route without changing it:

Parcel	Cost returned	Energy before	Expected after	Actual after
map	2	16	14	13
lamp	5	13	8	7
key	2	7	5	4

For the code shown here, the first divergence occurs immediately after the correct delivery cost is applied: an extra `energy -= 1` charges an undocumented fee.

Repair the earliest wrong transition:

PYTHON

```
def run_route(parcels, starting_energy):
    """Return remaining energy after delivering every parcel."""
    energy = starting_energy
    for parcel in parcels:
        cost = delivery_cost(parcel["distance"], parcel["fragile"])
        energy -= cost
    return energy
```

Calculate the oracle independently: delivery costs are `2`, `5`, and `2`, totaling `9`, so starting at `16` leaves `7`. If an issue report had claimed the answer should be `6`, this calculation and the per-parcel trace would challenge the report itself. Debugging includes verifying the oracle.

Use corrected evidence:

PYTHON

```
assert delivery_cost(1, False) == 2
assert delivery_cost(2, True) == 5
assert run_route([], 16) == 16
assert run_route([{"name": "map", "distance": 1, "fragile": False}], 16) == 14
assert run_route(parcels, 16) == 7
```

This is a realistic lesson: user reports, comments, and assertions can be wrong. When actual transitions consistently contradict the stated expectation, return to the contract and recalculate it independently before forcing code to match.

Checkpoint: trust evidence, verify the oracle

Choose evidence over the initial story

Decide which tool or conclusion matches the uncertainty.

1. The debugger shows correct per-parcel costs totaling 9 from starting energy 16, but the report expects 6. What should happen next?

- **A.** Add an unexplained subtraction to force 6
- **B.** Recalculate and verify the expected result against the contract
- **C.** Ignore all observed values
- **D.** Catch AssertionError and report success

2. A bug appears only for the parcel named `key`. What is a focused debugger stop condition?

- **A.** `parcel["name"] == "key"`
- **B.** `True` at every line in every library
- **C.** Mutate the parcel inside the condition
- **D.** Delete all other parcels before preserving the original case

3. What should you do with temporary `breakpoint()` calls after the source repair is verified?

- **A.** Add one to every function
- **B.** Commit them without explanation
- **C.** Replace all assertions with them
- **D.** Remove them so normal runs do not halt unexpectedly

Write your choices before checking the answer key at the end of this book.

7. A minimal reproduction is small and complete

Imagine the real courier application has menus, colors, files, twenty parcel fields, and logging. The failure only depends on distance, fragility, starting energy, and the extra subtraction. A useful reproduction keeps those pieces:

```python
def delivery_cost(distance, fragile):
    cost = distance * 2
    if fragile:
        cost += 1
    return cost


def buggy_remaining_energy(distance, fragile, starting_energy):
    cost = delivery_cost(distance, fragile)
    energy = starting_energy - cost
    energy -= 1
    return energy


assert buggy_remaining_energy(1, False, 16) == 14
```

It is:

- **minimal enough** to expose one parcel and one extra transition;
- **complete** because every name, function, input, and expected value needed to run is included;
- **deterministic** because the same input produces the same failure;
- **safe to share** because it contains no secret paths, customer addresses, or access tokens.

A fragment such as `energy -= 1` is small but not complete. A full production repository may be complete but not minimal. Aim for the smallest standalone program that preserves the same behavior.

8. Reduce one dimension at a time

Start from a copied reproduction, not the only production artifact. Confirm it still fails, then reduce deliberately:

1. remove unrelated parcels;
2. remove unused mapping fields;
3. replace file or network input with a literal value of the same relevant shape;
4. inline a helper only if the failure remains;
5. remove formatting, UI, or logging;
6. rerun after **every** reduction;
7. restore the last removed element if the failure disappears.

Track results:

Reduction	Prediction	Still fails?	Conclusion
keep one map parcel	extra charge should remain	yes	multiple parcels are unnecessary
remove `name`	calculation should remain	yes	name is irrelevant to this failure
replace `fragile` with `False`	extra charge should remain	yes	fragile branch is not required
remove second subtraction	failure should disappear	no	subtraction is causally required

The last row is both a reduction and a controlled experiment. It connects one line to the observed mismatch.

Do not reduce away the environment too early

If a bug depends on Python version, operating system, package version, working directory, locale, timing, or call order, that detail is part of the reproduction. Minimal means no irrelevant pieces, not no context. :::

9. Write a bug report someone else can run

A useful report includes:

```text
Title: Courier charges an extra energy point per parcel

Environment:
- Python version:
- operating system/editor if relevant:
- clean command used to run:

Steps:
1. Save the attached complete example as courier_minimal.py.
2. Run `python courier_minimal.py`.

Expected:
One non-fragile distance-1 delivery costs 2; 16 energy should leave 14.

Actual:
The assertion fails because the function returns 13.

First divergence:
Cost is correctly 2. Energy becomes 14, then an additional subtraction makes 13.

Attachment:
Minimal complete example with synthetic, non-sensitive data.
```

Include the exact traceback for an exception or `repr` of surprising values. Do not paraphrase `TypeError` as “it crashed” or silently edit the message.

Sanitize without changing shape. Replace a real path with `/example/project/data.txt`, a token with `<redacted-token>`, and personal data with synthetic records. Rerun the sanitized reproduction to prove it still fails.

10. Run a debugger-to-reproduction lab

The courier gets a new rule: fragile parcels cost one extra point only when distance is greater than one. The implementation applies the fee to every fragile parcel:

```python
def delivery_cost(distance, fragile):
    cost = distance * 2
    if fragile:
        cost += 1
    return cost


route = [
    {"name": "glass key", "distance": 1, "fragile": True, "color": "blue"},
    {"name": "lamp", "distance": 3, "fragile": True, "color": "gold"},
    {"name": "map", "distance": 4, "fragile": False, "color": "green"},
]
```

Complete the lab:

1. write expected costs for all three parcels;
2. save a complete local script and reproduce the wrong first cost;
3. pause before `cost += 1` and inspect `distance` and `fragile`;
4. use a condition to pause only when `distance == 1`;
5. step over the fee and record before/after state;
6. reduce to one function call and one assertion;

7 remove `name`, `color`, and the other parcels one at a time, rerunning each time;

8 state a falsifiable hypothesis;

9 repair the condition and add checks for both sides of the boundary;

10 remove temporary breakpoints and run the saved script normally.

Target regression checks:

```python
assert delivery_cost(1, False) == 2
assert delivery_cost(1, True) == 2
assert delivery_cost(2, True) == 5
assert delivery_cost(3, True) == 7
```

Do not copy those expected values blindly. Explain each from the updated rule.

Show the minimal repair after finishing the debugger trace

The fee needs both conditions:

```python
def delivery_cost(distance, fragile):
    """Return energy spent under the distance-sensitive fragile rule."""
    cost = distance * 2
    if fragile and distance > 1:
        cost += 1
    return cost
```

The one-parcel failure `delivery_cost(1, True) == 2` is the most focused regression. The distance-two and non-fragile checks protect the neighboring branches. Remove debugger stops, save the script, and run all four checks in a normal process before comparing your result.

Checkpoint: shrink without losing the cause

Build a runnable bug report

Choose what makes a reproduction useful to another investigator.

1. Which description best fits a minimal complete reproduction?

- **A.** One suspicious line with all names undefined
- **B.** The entire production system and private database
- **C.** The smallest standalone code, input, and expected behavior that still trigger the same failure
- **D.** A screenshot saying "broken"

2. Why rerun after every removal while reducing a case?

- **A.** To detect the exact reduction that removed the failure
- **B.** To guarantee the environment is irrelevant
- **C.** To make the file longer
- **D.** To avoid stating an expected result

3. What should happen after replacing sensitive values with synthetic ones?

- **A.** Assume the bug remains without running

- **B.** Publish the original secrets too
- **C.** Delete the expected result
- **D.** Rerun the sanitized example to confirm it preserves the failure

Write your choices before checking the answer key at the end of this book.

11. Key points for the arcade challenge

- Choose the simplest tool that can answer the current uncertainty.
- A breakpoint normally pauses before its highlighted line; inspect, step once, and compare the after-state with a predicted transition.
- Use frames to understand caller and callee state, and conditional breakpoints to reach the relevant iteration.
- `breakpoint()` and `python -m pdb script.py` are useful local options; remove temporary stops after the investigation.
- Verify the oracle as carefully as the implementation.
- A minimal reproduction must be small, complete, deterministic, safe, and independently runnable.
- Reduce one dimension at a time and rerun after every change.

References and next steps

- Python documentation: `pdb` — The Python Debugger
- Python documentation: built-in `breakpoint()`
- VS Code documentation: Python debugging
- VS Code Python tutorial: run and debug

The unit challenge supplies a small arcade with several independent defects. Progressive assertions and a hint ladder will help you isolate parsing, movement, and exception-boundary behavior without hiding unexpected failures.

Unit Challenge: Repair the Glitched Arcade

Online lesson and runnable notebook

Repair Pixel's command parser, movement rules, and exception boundary so valid moves reach the portal while rejected commands keep useful evidence.

- **Level:** Python Foundations · Unit 8 challenge
- **Estimated time:** 60–120 minutes
- **Outcome assessed:** Classify failure evidence, repair exception design and movement logic, and verify one cause at a time without hiding unexpected defects.
- **Evidence:** Unchanged progressive assertions, a portal-opening round, three contextual rejection messages, one debugging record, and a clean rerun

1. Get Pixel to the portal without hiding the glitches

An old arcade cabinet controls **Pixel**, a tiny character stranded behind a glitchy joystick. Every accepted command moves Pixel on a coordinate grid. The portal opens only when the final position is exactly `(2, 1)`.

The cabinet has this command tape:

```python
commands = [
    "UP 2",
    "RIGHT 3",
    "LEFT many",
    "JUMP 1",
    "DOWN 1",
    "LEFT 1",
    "RIGHT 0",
]

portal = (2, 1)
```

Some commands are valid moves. Others should be rejected with the turn number and a useful reason. The supplied implementation runs, but the portal stays dark. Your job is to use the staged evidence to repair the command boundary and the silent movement bug.

At the end, the game should report:

```text
Pixel stopped at (2, 1).
Rejected commands: 3
Portal opened!
```

You have a repair manual, not an answer key

Work on one subsystem at a time. Run the closest checks, state a hypothesis, make one change, and rerun earlier checks. Open hints only after you can name the exact assertion, exception, or value blocking you. :::

2. Read the command and movement rules

Commands

- A command is a string containing exactly two whitespace-separated parts.
- The first part is a direction. Input is case-insensitive and normalizes to `UP`, `RIGHT`, `DOWN`, or `LEFT`.
- The second part must convert to a **positive** integer. Zero is not positive.
- Every rejected command raises a contextual `ValueError` containing its 1-based turn number.
- When unpacking or integer conversion raises the original `ValueError`, the contextual error preserves it as `__cause__` with `raise ... from error`.
- Unsupported direction and non-positive steps are explicit rule failures, so they have no invented lower-level cause.

Movement

Pixel begins at `(0, 0)`. Coordinates use `(x, y)`:

Direction	Coordinate change	Example from `(3, 2)`
`UP`	add steps to y	`UP 2` → `(3, 4)`
`RIGHT`	add steps to x	`RIGHT 2` → `(5, 2)`
`DOWN`	subtract steps from y	`DOWN 2` → `(3, 0)`
`LEFT`	subtract steps from x	`LEFT 2` → `(1, 2)`

`move_pixel` receives a validated command and returns a **new tuple**. It does not mutate the supplied position.

Round recovery

`play_round` parses each command in encounter order. It catches only an anticipated parser `ValueError`, records the message, and continues to the next turn. Movement happens outside that narrow `try` block. If movement code has an unexpected defect, that defect must remain visible instead of being called bad player input.

For the supplied tape:

- turns 3, 4, and 7 are rejected;
- the four accepted moves are `UP 2`, `RIGHT 3`, `DOWN 1`, and `LEFT 1`;
- accepted moves lead `(0, 0)` → `(0, 2)` → `(3, 2)` → `(3, 1)` → `(2, 1)`; and
- the input list and strings remain unchanged.

3. Start from the contract

Run this starter unchanged. It is valid Python and intentionally looks plausible. Let the assertions reveal which promises it violates before editing. Preserve the three public names, signatures, and docstrings.

```python
VALID_DIRECTIONS = {"UP", "RIGHT", "DOWN", "LEFT"}


def parse_command(text, turn):
    """Return a normalized (direction, steps) tuple or raise contextual ValueError."""
    try:
        direction, raw_steps = text.split()
    except ValueError:
        raise ValueError(f"turn {turn}: command needs exactly two parts")

    direction = direction.upper()
    if direction not in VALID_DIRECTIONS:
        raise ValueError(f"turn {turn}: unsupported direction {direction!r}")

    try:
        steps = int(raw_steps)
    except ValueError:
        raise ValueError(f"turn {turn}: steps must be an integer")

    if steps < 0:
        raise ValueError(f"turn {turn}: steps must be positive")
    return direction, steps


def move_pixel(position, command):
    """Return Pixel's new position after one validated command."""
    x, y = position
    direction, steps = command

    if direction == "UP":
        y += steps
    elif direction == "RIGHT":
        x += steps
    elif direction == "DOWN":
        y -= steps
    elif direction == "LEFT":
        x += steps
    return x, y


def play_round(commands, portal):
    """Return final position, rejected messages, and whether the portal opened."""
    position = (0, 0)
    rejected = []

    for turn, text in enumerate(commands, start=1):
        try:
            command = parse_command(text, turn)
            position = move_pixel(position, command)
        except Exception as error:
            rejected.append(str(error))

    return position, rejected, position == portal
```

Do not list possible repairs yet. First run the parser checks in Section 5 and keep the first failure. A failed check is a map to one contract boundary.

4. Isolate one arcade subsystem at a time

Treat the arcade as three cooperating systems:

Subsystem	Input	Normal result	Anticipated failure	Focused evidence
parser	text and turn	normalized tuple	contextual `ValueError`	return value, message, `__cause__`
movement	position and validated command	new position	none for valid commands	before/after coordinates
round runner	command sequence and portal	final tuple, messages, opened flag	parser rejection and continuation	accepted path, rejection count, final state

Follow this order:

1. **Normalization:** make valid lower/mixed-case commands return uppercase directions and integer steps.
2. **Command shape and conversion:** inspect the new error and its original cause separately.
3. **Domain validation:** reject unsupported directions, zero, and negatives.
4. **Coordinates:** verify one direction at a time from the same starting point.
5. **Minimal regression:** reduce the wrong round to one `LEFT` move.
6. **Recovery boundary:** parse inside the protected region, then move outside it.
7. **Integration:** run the supplied tape and confirm final position, rejection evidence, input ownership, and portal state.

For one defect, use this investigation loop:

```text
Reproduce -> copy exact evidence -> state one cause -> change one line or boundary
          -> rerun the focused check -> rerun all earlier checks
```

Do not edit the expected values to create green output

If an assertion surprises you, compare it with the written command or movement rule. Repair an incorrect oracle only when you can demonstrate that the written contract contradicts it. :::

5. Run progressive assertions

Run each stage after the previous stage passes. The helper below captures the specific exception and fails clearly when no `ValueError` is raised:

```python
def captured_value_error(action):
    """Run action and return its ValueError, or fail if none is raised."""
    try:
        action()
    except ValueError as error:
        return error
    raise AssertionError("expected ValueError")
```

Stage A: normalize valid commands

```python
assert parse_command("UP 2", 1) == ("UP", 2)
assert parse_command("left 3", 2) == ("LEFT", 3)
assert parse_command("  Right   4  ", 3) == ("RIGHT", 4)
assert parse_command("dOwN 1", 4) == ("DOWN", 1)
```

Stage B: preserve a malformed-shape cause

Unpacking one or three parts raises the lower-level `ValueError`. Add turn context without losing that cause:

```python
shape_error = captured_value_error(lambda: parse_command("UP", 5))
assert "turn 5" in str(shape_error)
assert "two" in str(shape_error) or "parts" in str(shape_error)
assert isinstance(shape_error.__cause__, ValueError)

extra_part_error = captured_value_error(lambda: parse_command("UP 2 NOW", 6))
assert "turn 6" in str(extra_part_error)
assert isinstance(extra_part_error.__cause__, ValueError)
```

Stage C: preserve an integer-conversion cause

```python
conversion_error = captured_value_error(lambda: parse_command("LEFT many", 3))
assert "turn 3" in str(conversion_error)
assert "integer" in str(conversion_error)
assert isinstance(conversion_error.__cause__, ValueError)
assert "many" in str(conversion_error.__cause__)
```

Stage D: reject unsupported and non-positive moves

These are direct contract decisions, not translated lower-level failures:

```python
direction_error = captured_value_error(lambda: parse_command("JUMP 1", 4))
assert "turn 4" in str(direction_error)
assert "JUMP" in str(direction_error)
assert direction_error.__cause__ is None

zero_error = captured_value_error(lambda: parse_command("RIGHT 0", 7))
assert "positive" in str(zero_error)
assert zero_error.__cause__ is None

negative_error = captured_value_error(lambda: parse_command("DOWN -2", 8))
assert "turn 8" in str(negative_error)
assert "positive" in str(negative_error)
```

Stage E: move in all four directions

```python
origin = (3, 2)
assert move_pixel(origin, ("UP", 2)) == (3, 4)
assert move_pixel(origin, ("RIGHT", 2)) == (5, 2)
assert move_pixel(origin, ("DOWN", 2)) == (3, 0)
assert move_pixel(origin, ("LEFT", 2)) == (1, 2)
assert origin == (3, 2)
```

Keep this smallest regression check even after the full round passes:

```python
assert move_pixel((3, 1), ("LEFT", 1)) == (2, 1)
```

Stage F: reject bad commands and keep playing

```python
commands_before = commands.copy()
position, rejected, opened = play_round(commands, portal)

assert position == (2, 1)
assert len(rejected) == 3
assert "turn 3" in rejected[0]
assert "turn 4" in rejected[1]
assert "turn 7" in rejected[2]
assert opened is True
assert commands == commands_before
assert portal == (2, 1)
```

Confirm the round boundary does not hide a movement defect. Temporarily replace `move_pixel` with a version that raises `RuntimeError`, call `play_round`, and observe that the failure propagates. Restore the real function before the full rerun. Do not add a `RuntimeError` handler merely to make this experiment quiet.

Finally, create a short command list of your own. Predict every accepted move, every rejection, and the final position on paper, then add assertions for its result without changing any supplied expected values.

6. Use the hint ladder only when needed

Hint 1

Let the failing assertion identify the subsystem. `__cause__` belongs to parser translation, a wrong coordinate belongs to movement, and a swallowed `RuntimeError` belongs to the round's `try` boundary. Do not repair all three at once.

Hint 2

When Python itself raises during unpacking or `int`, retain that exception after adding turn context. "Positive" excludes zero. `LEFT` and `RIGHT` must use opposite x-coordinate operations. The round can catch parser `ValueError` without placing movement in the same protected block.

Hint 3

Fill only the missing decisions in these focused shapes:

```python
try:
    direction, raw_steps = text.split()
except ValueError as error:
    raise ValueError(...) from error

if steps <= 0:  # Replace the comparison if your evidence supports another one.
    raise ValueError(...)

elif direction == "LEFT":
    x = ...  # Replace the value with the intended coordinate update.

try:
    command = parse_command(text, turn)
except ValueError as error:
```

PYTHON (CONTINUED)

```
    rejected.append(...)
else:
    position = move_pixel(position, command)
```

7. Keep debugging evidence

Preserve one failure that changed your understanding. The one-move `LEFT` check is a good candidate because it removes parsing, the loop, and the portal while retaining the coordinate defect.

Reproduction	Actual evidence	Hypothesis	One change	Focused rerun	Full rerun	Regression check
exact call/check	value, message, or cause	one mechanism that could be false	one line or boundary	nearest check result	all earlier stages	smallest permanent check

Example evidence should be specific:

TEXT

```
Reproduction: move_pixel((3, 1), ("LEFT", 1))
Expected: (2, 1)
Actual: (4, 1)
Hypothesis: LEFT uses addition, the same x update as RIGHT.
Prediction: changing only LEFT to subtraction returns (2, 1); other directions stay unchanged.
```

After every repair, restart the notebook runtime and run from the starter or your final definitions through all progressive assertions. A portal that opens only because an old function remains in memory is not reliable evidence.

8. Compare with a complete solution

Show the complete arcade repair after attempting every stage

PYTHON

```
VALID_DIRECTIONS = {"UP", "RIGHT", "DOWN", "LEFT"}


def parse_command(text, turn):
    """Return a normalized (direction, steps) tuple or raise contextual ValueError."""
    try:
        direction, raw_steps = text.split()
    except ValueError as error:
        raise ValueError(
            f"turn {turn}: command needs exactly two parts"
        ) from error

    direction = direction.upper()
    if direction not in VALID_DIRECTIONS:
        raise ValueError(
            f"turn {turn}: unsupported direction {direction!r}"
        )

    try:
        steps = int(raw_steps)
    except ValueError as error:
        raise ValueError(
            f"turn {turn}: steps must be an integer; got {raw_steps!r}"
        ) from error
```

PYTHON (CONTINUED)

```
    if steps <= 0:
        raise ValueError(
            f"turn {turn}: steps must be positive; got {steps}"
        )
    return direction, steps


def move_pixel(position, command):
    """Return Pixel's new position after one validated command."""
    x, y = position
    direction, steps = command

    if direction == "UP":
        y += steps
    elif direction == "RIGHT":
        x += steps
    elif direction == "DOWN":
        y -= steps
    elif direction == "LEFT":
        x -= steps
    else:
        raise ValueError(f"unvalidated direction reached movement: {direction!r}")
    return x, y


def play_round(commands, portal):
    """Return final position, rejected messages, and whether the portal opened."""
    position = (0, 0)
    rejected = []

    for turn, text in enumerate(commands, start=1):
        try:
            command = parse_command(text, turn)
        except ValueError as error:
            rejected.append(str(error))
        else:
            position = move_pixel(position, command)

    return position, rejected, position == portal
```

Why the repaired boundaries matter:

- unpacking and integer conversion can fail at a lower level, so chaining keeps both Python’s cause and the command’s turn context;
- unsupported direction and non-positive steps are direct parser rules, so no unrelated cause is manufactured;
- `LEFT` subtracts from x and the tuple returned is new;
- only parsing sits inside the round’s `try`, so an unexpected movement failure still produces honest developer evidence; and
- the portal result derives from the final position rather than being hard-coded.

Run every assertion from Section 5 unchanged after loading this solution.

9. Predict a changed arcade rule

The cabinet designer proposes two changes:

1 `STAY 1` becomes a supported command that leaves Pixel in place; and

2 the portal opens if Pixel finishes at Manhattan distance at most one from its center, not only on the exact coordinate.

Before editing:

- Which parser constant, movement branch, docstrings, and assertions change?
- Should `STAY 0` become valid, or does the positive-step rule still reject it?
- For portal `(2, 1)`, which neighboring coordinates now open it?
- Which behavior belongs in a small helper instead of making `play_round` harder to read?
- Which old assertions should remain unchanged as regression evidence?

Implement the changed rule only after writing examples for exact, adjacent, and two-steps-away positions. Do not weaken command validation accidentally while adding a new direction.

10. Check your understanding

Glitched arcade repair check

Connect each answer to the supplied contract and progressive evidence.

1. Why should `LEFT many` use `raise ... from error` after `int` fails?

- **A.** It makes `many` a valid integer
- **B.** It adds turn context while preserving the original conversion failure
- **C.** It skips every later command
- **D.** It hides the traceback

2. Why must `RIGHT 0` be rejected?

- **A.** Zero is text
- **B.** RIGHT is unsupported
- **C.** The contract requires a positive integer, and zero is not positive
- **D.** Every even number is invalid

3. What does `move_pixel((3, 1), ("LEFT", 1)) == (2, 1)` isolate?

- **A.** The LEFT coordinate rule without parsing or round recovery
- **B.** Every possible arcade feature
- **C.** Only integer conversion
- **D.** The browser’s progress storage

4. Why does `play_round` catch parser `ValueError` but move outside the `try` block?

- **A.** Movement can never fail
- **B.** A broader block runs faster
- **C.** It converts every defect into a portal success
- **D.** Expected command mistakes can be recorded, while unexpected movement defects remain visible

5. Which evidence most strongly supports the final repair?

- **A.** The portal opened once before the last edit
- **B.** From a clean run, all unchanged staged checks pass, three turns are rejected, and Pixel ends at `(2, 1)`
- **C.** Expected values were changed to match output
- **D.** Unexpected exceptions no longer appear because all are caught

Write your choices before checking the answer key at the end of this book.

11. Decide whether the challenge is complete

Evidence rubric

Evidence	Ready to record when
Parser	Normalization, shape, conversion, direction, and positive-step checks pass unchanged.
Movement	Four directions and the minimal LEFT regression behave as specified without input mutation.
Boundary	Three anticipated command failures are recorded, play continues, and an unexpected movement defect would propagate.
Payoff	Pixel finishes at `(2, 1)` and the computed portal flag is true.
Reasoning	One debugging record connects exact evidence, one hypothesis, one controlled change, and focused/full reruns.
Reproducibility	The definitions and all assertions pass after a clean restart and top-to-bottom run.

This button stores a self-reported marker only in this browser. It does not submit work, grade it, verify identity, or issue a certificate.

Key points

- Progressive assertions turn a multi-defect program into parser, movement, and boundary investigations.
- Exception chaining preserves a lower-level cause while adding turn context.
- Narrow handling continues after anticipated command mistakes without hiding programmer defects.
- A minimal logic reproduction exposes direction state without needing the whole arcade round.
- The strongest finish is a clean rerun you can explain, not merely a glowing portal from stale notebook state.

Unit 9: Files, Paths, and External Data

Online lesson and runnable notebook

Locate files predictably, preserve text, translate external formats, validate records, and produce reproducible outputs with rejection evidence.

- **Level:** Python Foundations · Unit 9
- **Estimated time:** 22–28 hours including the challenge
- **Unit outcome:** Locate and traverse files predictably; read and write text with explicit resource, encoding, newline, and replacement policies; translate CSV, JSON, YAML, and binary data into validated Python values; and build a reproducible import pipeline that preserves accepted data, rejected evidence, and original inputs.
- **Practice in:** Generated Colab notebooks and temporary local workspaces created by Python

1. Welcome to the signal archive

A program often begins with values already in memory: a list of names, a price, or a dictionary. Useful programs also need to cross boundaries. They load a configuration, discover yesterday's reports, accept a spreadsheet export, save a game, or inspect an image header. Beyond each boundary is data that your program did not create and cannot simply trust.

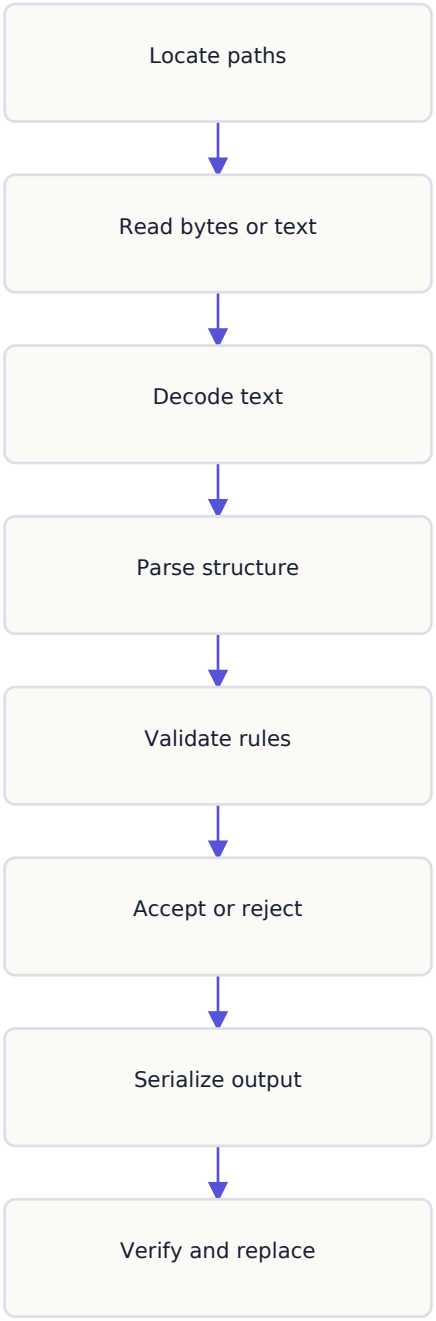
Imagine receiving a folder of starship signals. Before learning anything from it, your program must answer several practical questions:

- Where is the folder relative to the process that is running?
- Which entries are input files, and in what order should they be processed?
- Are the bytes UTF-8 text, text in another encoding, or not text at all?
- Does a valid CSV row also satisfy the archive's rules?
- Where should rejected rows and their reasons be recorded?
- Can a failed rerun leave the previous good archive untouched?

This unit gives each question enough room to become a usable skill. You will not merely call `read_text()` once. You will build and inspect complete, repeatable pipelines.

2. See the complete journey before studying each stage

External data becomes trustworthy only after several different responsibilities have succeeded; saving results crosses the same boundaries in reverse.



A parser can succeed while validation fails. A correct serializer can still write to the wrong path. Naming each stage helps you locate failures instead of blaming “the file” as one indivisible operation.

The stages have different contracts:

Stage	Question it answers	Typical evidence
Locate	Which filesystem object do we mean?	a `Path`, directory tree, sorted file list
Read	How do values enter the process, and who closes the resource?	mode, open handle, bytes or text

Stage	Question it answers	Typical evidence
Decode	How do byte values map to characters?	encoding name, newline policy, decode error
Parse	Does the text follow a format grammar?	CSV fields, JSON values, YAML values
Validate	Do those values obey our application rules?	accepted record or a precise reason
Serialize	How will Python values be represented outside Python?	deterministic text or bytes
Replace	When is an output complete enough to become official?	verified temporary artifact and final path

When an error occurs, ask which contract was broken. A missing path is not a Unicode problem. Malformed JSON is not the same as a valid JSON object missing a required key. Keeping the stages separate makes both code and explanations more precise.

3. Work in a disposable practice folder

File exercises should be safe to rerun. Hard-coded paths such as `/Users/sam/Desktop` work for one person and fail everywhere else. Writing practice output into the repository also leaves clutter and can accidentally replace real files.

Most lessons therefore use `TemporaryDirectory`:

PYTHON

```
from pathlib import Path
from tempfile import TemporaryDirectory

with TemporaryDirectory() as temporary_name:
    workspace = Path(temporary_name)
    inbox = workspace / "incoming"
    inbox.mkdir()

    note_path = inbox / "welcome.txt"
    note_path.write_text("Olá, archive!\n", encoding="utf-8")

    print(note_path.relative_to(workspace))
    print(note_path.read_text(encoding="utf-8"))
```

Python creates a unique directory, the example writes only inside it, and the directory is removed after the `with` block. Every learner gets the same small world regardless of operating system. Each larger lab creates all of its own input files, so you can restart the kernel and rerun it from the top.

Temporary means temporary

Move evidence you want to keep—such as a debugging note or final function—out of the temporary directory before its `with` block ends. The directory is a safe laboratory, not permanent storage. :::

A notebook and a local script may start with different **current working directories**. You will inspect that value rather than assume it, and you will build paths from a deliberate root. Later, when Unit 10 turns programs into projects, that root can come from configuration, a command-line argument, or a known package resource.

4. Take this route through the unit

Step	Lesson	Capability you will practice
1	Find, Read, and Write Files Reliably	Construct and inspect paths, traverse folders deterministically, manage file handles, stream lines, and choose writing modes without accidental truncation.
2	Keep Text Intact Across Encodings and Newlines	Distinguish characters from bytes, diagnose decoding failures, normalize comparisons carefully, and preserve multilingual text across newline conventions.
3	Move Records Between CSV, JSON, YAML, and Python	Choose a format, use its reader and writer correctly, make type conversion explicit, and separate parsed structure from trusted records.
4	Turn Messy Text into Trustworthy Records	Recognize and extract patterns, validate structural and domain rules, and preserve accepted values, rejected evidence, and source lineage.
5	Cross the Text–Binary Boundary Safely	Inspect bytes, use binary file modes and chunks, work with in-memory streams, and encode text only when a format defines the boundary.
6	Unit Challenge: Restore the Starship Signal Archive	Repair a multilingual, multi-file import pipeline and prove its discovery, validation, error, and replacement policies with progressive assertions.

Each lesson contains three short quiz checkpoints and a larger integration lab. Do not rush past a surprising output. Record what you predicted, what actually happened, and which boundary explains the difference.

5. Carry Unit 8’s debugging method into external data

Unit 8 taught you to reproduce a failure, gather evidence, state one hypothesis, and change one thing. External-data failures reward exactly that discipline. Preserve the smallest artifact that reproduces the problem:

```text
Path or source name:
Expected contract:
Observed value or exception:
Smallest input that reproduces it:
Stage where expectation first diverges:
```

For a decoding error, record the encoding you attempted and the offending byte position. For a rejected record, preserve the source file, logical record number, raw fields, and reason. For a wrong output, compare parsed Python values as well as serialized text. Evidence should explain *which* boundary failed.

Never experiment on the only copy

Keep original external inputs unchanged. Write derived output elsewhere, and do not replace a known-good artifact until the complete new artifact has been written and checked. The lessons use disposable fixtures, but the habit applies to real research data, customer exports, save files, and configurations. :::

6. Budget time for deliberate practice

Plan roughly four to six hours for each lesson and one to two hours for the challenge. That time includes:

1. predicting selected outputs before running them;
2. typing and tracing complete examples;
3. making the requested changes and observing the consequences;
4. diagnosing at least one intentional failure;
5. completing the lesson lab without relying on old notebook state; and
6. answering the quizzes from your reasoning, not from trial-and-error clicks.

Reading the page once is not completion. A more useful signal is whether you can create a fresh temporary workspace, explain every path and format decision, and rerun the pipeline with the same results.

7. Prepare for the archive restoration

At the end of the unit, you will inherit a plausible but defective signal archive program. It discovers CSV files, normalizes records, quarantines bad rows, and writes a JSON archive. Several choices are subtly unsafe: ordering is reversed, a pattern accepts extra characters, one priority slips through, an exception boundary is too broad, and the writer does not satisfy its output contract.

The challenge gives you function names, docstrings, fixtures, progressive assertions, and three hints. Your job is to investigate and repair the program, not invent an application from a blank page. A successful clean rerun will reveal the restored multilingual beacon while preserving evidence for every rejected signal.

Unit 10 will then use these boundary skills inside modules, environments, and projects. You will move from one reliable notebook pipeline to code that can be installed and reused. There is no final project or certificate in this release; both are planned, and we will announce them when they are ready.

Key points

- External data crosses several contracts: location, resources, decoding, parsing, validation, serialization, and replacement.
- A temporary workspace makes file labs portable, self-contained, and safe to rerun.
- Original inputs, rejected evidence, and unexpected exceptions should remain visible.

- Deterministic discovery and output make a pipeline easier to reproduce and debug.
- Unit 8’s evidence loop remains the right response when a file or record behaves unexpectedly.

Find, Read, and Write Files Reliably

Online lesson and runnable notebook

Build predictable paths, inspect and traverse directories, manage text-file resources, and write without surprising truncation.

- **Level:** Python Foundations
- **Estimated time:** 4–5 hours
- **You will learn:** Build and inspect paths, distinguish files from directories, traverse a workspace in a stable order, manage file handles, choose between whole-file and streaming reads, and select a safe writing mode.
- **Practice in:** Google Colab, JupyterLab, or a local editor using temporary workspaces

A wildlife team returns from the field with notes from several observation stations. Your task is to find the notes, read them without leaking resources, and create a daily summary. That sounds simple, but the first version can fail by starting in a different folder, including yesterday's generated report as new input, reading a directory as though it were a file, or erasing an existing log with the wrong mode.

Keep these questions beside your notebook:

- What root directory is this path relative to?
- Does the path exist, and is it the kind of object the operation needs?
- Who closes the file if reading or processing fails?
- Is the file small enough to read at once, or should records be streamed?
- Does writing replace, append, or require a new destination?
- Is traversal order part of the observable result?

1. Paths describe locations; files and directories occupy them

A filesystem organizes named entries inside directories. A **path** is a route to an entry, not the file's contents. `pathlib.Path` represents that route with rules appropriate to the operating system.

PYTHON

```
from pathlib import Path

report_path = Path("field-notes") / "north" / "sightings.txt"

print(report_path)
print(report_path.name)
print(report_path.stem)
print(report_path.suffix)
print(report_path.parent)
print(report_path.parts)
```

The `/` operator here joins path components; it does not divide numbers. For the last component `sightings.txt`:

- `.name` is `"sightings.txt"`;
- `.stem` is `"sightings"`;
- `.suffix` is `".txt"`;
- `.parent` is the path ending in `north`; and
- `.parts` exposes every component as a tuple.

A suffix is only part of a name. Renaming a file from `.txt` to `.json` does not convert its contents to JSON. Formats require parsing and serialization, which you will practice in Lesson 3.

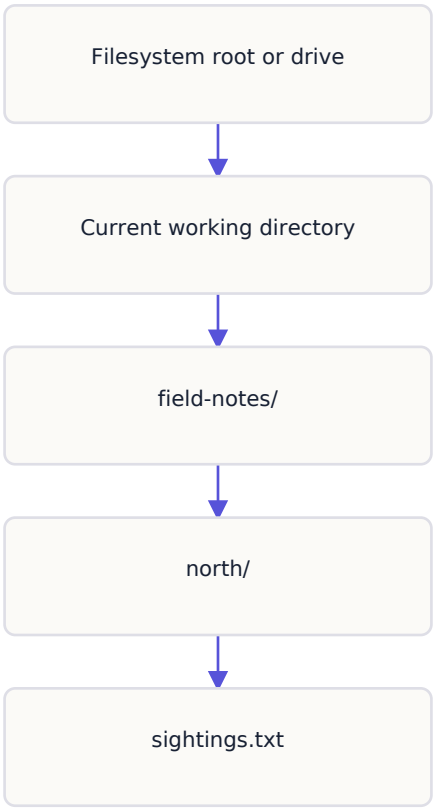
Path construction does not create or open anything:

```python
imagined = Path("archive") / "2026" / "observations.tar.gz"
print(bool(imagined))
print(imagined.exists())
print(imagined.name, imagined.stem, imagined.suffix, imagined.suffixes)
```

`bool(imagined)` is `True` even when the path is missing; a `Path` is an object, not a filesystem-state test. Notice also that `.stem` removes only the final suffix (`.gz`), while `.suffixes` reports `['.tar', '.gz']`. Names with several dots need a documented naming rule rather than repeated string slicing.

Do not join components with `str(root) + "/north/sightings.txt"`. Separators, drives, and absolute components vary by platform. Let `Path` perform path composition, and convert to text only when an external API actually requires a string.

A relative path begins at the process's current working directory; an absolute path begins at a filesystem anchor such as a root or drive.



If the current working directory changes, the same relative spelling can refer to a different location. The `Path` object does not search for the intended project automatically.

Inspect rather than assume:

```python
from pathlib import Path

working_directory = Path.cwd()
candidate = Path("field-notes") / "north" / "sightings.txt"

print("working directory:", working_directory)
print("relative?:", not candidate.is_absolute())
print("candidate from cwd:", working_directory / candidate)
```

`Path.resolve()` can produce an absolute path and resolve symbolic components, but its exact filesystem behavior depends on the `strict` argument and the platform. It does not prove that you selected the correct project root. The best repair for an ambiguous relative path is usually to pass the intended root into the function.

```python
def notes_path(workspace, station_name):
    """Return the notes path for one station under an explicit workspace."""
    return Path(workspace) / "field-notes" / station_name / "sightings.txt"
```

The function no longer depends silently on wherever the process happened to start.

2. Create a safe field-notes workspace

The next fixture creates every file it needs. Run the cell again whenever you want a clean starting point.

```python
from pathlib import Path
from tempfile import TemporaryDirectory

temporary_directory = TemporaryDirectory()
workspace = Path(temporary_directory.name)
notes_root = workspace / "field-notes"
(notes_root / "north").mkdir(parents=True)
(notes_root / "south").mkdir(parents=True)
(notes_root / "empty-station").mkdir()

(notes_root / "north" / "sightings.txt").write_text(
    "otter,3\nheron,2\n",
    encoding="utf-8",
)
(notes_root / "south" / "sightings.txt").write_text(
    "fox,1\notter,2\n",
    encoding="utf-8",
)
(notes_root / "README.md").write_text(
    "Field notes are recorded once per station.\n",
    encoding="utf-8",
)

print("workspace:", workspace)
```

`parents=True` permits Python to create missing parents. Without it, creating `field-notes/north` would fail if `field-notes` did not exist. By default, `mkdir()` also fails if the final directory already exists.

`exist_ok=True` is appropriate only when an existing directory satisfies the contract; do not use it to silence a collision that ought to be investigated.

Ask what each path currently represents:

```python
north_note = notes_root / "north" / "sightings.txt"
missing_note = notes_root / "west" / "sightings.txt"

for path in [notes_root, north_note, missing_note]:
    print(
        path.relative_to(workspace),
        "exists=", path.exists(),
        "file=", path.is_file(),
        "directory=", path.is_dir(),
    )
```

`exists()` alone is not enough. A directory exists, but `read_text()` requires a file. A path can also disappear between a check and the following operation, so handle the operation's exception at a boundary where absence is genuinely expected.

Checkpoint: make location explicit

Reason about paths before opening them

Choose the statement that follows from the path contract.

1. What determines the starting point of `Path("reports/today.txt")`?

- **A.** The location of the last file opened
- **B.** The process's current working directory
- **C.** The learner's Downloads directory
- **D.** The file suffix

2. Why check `is_file()` rather than only `exists()` before a read that requires a regular file?

- **A.** Because files never exist
- **B.** Because `exists()` changes the path
- **C.** Because a directory or another filesystem entry can exist at that path
- **D.** Because `is_file()` decodes UTF-8

3. Which design is least dependent on notebook launch location?

- **A.** Repeatedly call `Path.cwd()` inside every helper
- **B.** Pass a workspace root into the function and join paths beneath it
- **C.** Guess that the current directory is always the repository
- **D.** Search the entire computer for a matching filename

Write your choices before checking the answer key at the end of this book.

3. Discover inputs in a deterministic order

`iterdir()` yields a directory's immediate children. `glob()` selects paths that match a pattern. `rglob()` searches recursively.

PYTHON
```
print("immediate entries:")
for path in sorted(notes_root.iterdir(), key=lambda item: item.name):
    print("-", path.name)

print("station notes:")
station_files = sorted(notes_root.glob("*/sightings.txt"))
for path in station_files:
    print("-", path.relative_to(notes_root))
```

Do not rely on the order returned by the filesystem. Sort when order affects summaries, identifiers, tests, or reproducibility. Sorting `Path` objects is usually enough for paths with a shared root; an explicit key documents another policy such as case-insensitive names.

Pattern matching should describe intended inputs, not merely all text files. This broad search can accidentally include generated output on the next run:

PYTHON
```
output_path = notes_root / "daily-summary.txt"
output_path.write_text("generated output\n", encoding="utf-8")

broad_matches = sorted(notes_root.rglob("*.txt"))
print([path.relative_to(notes_root) for path in broad_matches])
```

The pattern `*/sightings.txt` encodes two useful constraints: one station directory and the exact input filename. You can also filter explicitly:

PYTHON
```
input_files = [
    path
    for path in sorted(notes_root.rglob("*.txt"))
    if path.name == "sightings.txt" and path != output_path
]
```

Choose a rule that remains correct after output exists. "Delete the output before discovery" is fragile because a crash can leave it behind.

4. A file handle is a managed, stateful resource

`Path.read_text()` is convenient for a small file. Underneath, Python opens a text stream, reads it, and closes it. When you need line-by-line processing or more control, open the file explicitly with a `with` statement:

PYTHON
```
first_note = station_files[0]

with first_note.open(mode="r", encoding="utf-8") as handle:
    print("inside block, closed?:", handle.closed)
    first_line = handle.readline()
    remaining_text = handle.read()

print("outside block, closed?:", handle.closed)
print("first line:", repr(first_line))
print("remaining:", repr(remaining_text))
```

A handle has a **cursor**. `readline()` advances past one line; the following `read()` begins from that position, not from the start. Opening the file again creates a new handle with a new cursor.

Make cursor exhaustion observable:

```
PYTHON
with first_note.open(encoding="utf-8") as handle:
    first_pass = handle.read()
    second_pass = handle.read()

print("first pass:", repr(first_pass))
print("second pass:", repr(second_pass))
assert second_pass == ""
```

You could call `handle.seek(0)` before the second read, but reopening is often clearer when you truly need two independent passes. The built-in form `open(first_note, encoding="utf-8")` also accepts `Path` objects; `Path.open()` is used here because it keeps the location and operation visually together.

The context manager closes the resource even when code inside the block raises an exception. It does not suppress that exception:

```
PYTHON
try:
    with first_note.open(encoding="utf-8") as handle:
        raw_count = handle.readline().split(",")[1]
        int("not-a-number")
except ValueError as error:
    print(type(error).__name__, error)

print("closed after failure?:", handle.closed)
```

That separation is valuable: resource cleanup is guaranteed, while the unexpected or deliberately caught failure remains observable.

A closed handle cannot read more data

Store the values you need inside the `with` block. Returning a handle from a helper whose context has already ended returns a closed resource, not deferred file contents. :::

5. Read small files whole; stream large or record-oriented files

For a small configuration or note, whole-file methods are clear:

```
PYTHON
text = first_note.read_text(encoding="utf-8")
lines = text.splitlines()
print(lines)
```

`read_text()` returns one string. It is then possible to split, search, or parse that value. The complete contents must fit comfortably in memory.

For a large log or naturally line-oriented input, iterate over the handle:

```
PYTHON
def count_animals(path):
    """Return total observed animals from a small comma-separated note."""
    total = 0
    with Path(path).open(encoding="utf-8") as handle:
```

PYTHON (CONTINUED)

```
        for line_number, raw_line in enumerate(handle, start=1):
            line = raw_line.rstrip("\n")
            animal, raw_count = line.split(",")
            try:
                total += int(raw_count)
            except ValueError as error:
                raise ValueError(
                    f"{path}:{line_number}: invalid count {raw_count!r}"
                ) from error
    return total


for path in station_files:
    print(path.parent.name, count_animals(path))
```

Only one line at a time needs to be held by the loop. `enumerate` preserves line context for diagnostics. The narrow `rstrip("\n")` removes the line terminator without deleting meaningful leading or trailing spaces. Lesson 2 develops newline policies further, and Lesson 3 replaces the deliberately simple comma split with the CSV parser needed for quoted fields.

Choose from the data shape and size, not from a rule that one method is always superior.

Checkpoint: manage reads and discovery

Choose a reliable reading boundary

Reason about ordering, handles, and cursors.

1. A combined report must be identical across reruns. What should discovery do?

- **A.** Trust whatever order the filesystem returns
- **B.** Open only the shortest filename
- **C.** Shuffle paths before every run
- **D.** Select intended inputs and sort them by a documented key

2. After `readline()` consumes the first line, what does `read()` on the same handle return?

- **A.** The entire file from the beginning
- **B.** The remaining text from the current cursor position
- **C.** Only the filename
- **D.** A new handle

3. What does a file `with` statement guarantee when processing raises an exception?

- **A.** The handle is closed while the exception remains available
- **B.** Every exception is silently ignored
- **C.** The file contents are automatically repaired
- **D.** The cursor returns to zero

Write your choices before checking the answer key at the end of this book.

6. Writing modes express different promises

The common text modes are:

Mode	Promise	Important risk or behavior
`"r"`	read an existing file	missing path raises `FileNotFoundError`
`"w"`	create or replace a file	an existing file is truncated when opened
`"a"`	append at the end, creating if needed	reruns may duplicate content
`"x"`	create a new file only	existing path raises `FileExistsError`

`write()` does not add a newline:

PYTHON
```
mode_demo = workspace / "mode-demo.txt"
with mode_demo.open("w", encoding="utf-8") as handle:
    handle.write("north")
    handle.write("south")

print(repr(mode_demo.read_text(encoding="utf-8")))
```

`write()` returns the number of text characters accepted by the stream. The return value can support a focused check, although reopening and comparing the artifact is stronger evidence of the saved result:

PYTHON
```
with mode_demo.open("a", encoding="utf-8") as handle:
    character_count = handle.write("\nnight shift\n")

assert character_count == len("\nnight shift\n")
assert mode_demo.read_text(encoding="utf-8").endswith("night shift\n")
```

Run that cell twice and the line appears twice. Append mode preserves existing content, but it does not make retries idempotent.

The result is `'northsouth'`. Include the separator required by the file contract:

PYTHON
```
with mode_demo.open("w", encoding="utf-8", newline="\n") as handle:
    handle.write("north\n")
    handle.write("south\n")
```

Opening with `"w"` immediately truncates an existing file, even if a later calculation fails before the first `write`. Compute and validate the complete text before opening the destination when practical:

PYTHON
```
summary_lines = []
for path in station_files:
    summary_lines.append(f"{path.parent.name}: {count_animals(path)}")

summary_text = "\n".join(summary_lines) + "\n"
assert summary_text.count("\n") == len(summary_lines)
output_path.write_text(summary_text, encoding="utf-8", newline="\n")
print(summary_text)
```

Appending is suitable for an event log only when duplicate appends on retry are acceptable or prevented by another rule. Exclusive creation is useful when replacing an existing destination would indicate a mistake:

PYTHON
```
reservation = workspace / "reserved-name.txt"
reservation.write_text("first owner\n", encoding="utf-8")

try:
    with reservation.open("x", encoding="utf-8") as handle:
        handle.write("second owner\n")
except FileExistsError:
    print("destination already exists; original preserved")
```

Lesson 2 will build a temporary-write, verify, and replace sequence for outputs that should become official only when complete.

7. Handle expected filesystem failures at the right boundary

Common filesystem exceptions carry different evidence:

- `FileNotFoundError`: a required path or parent is missing;
- `FileExistsError`: exclusive creation or directory creation collided;
- `IsADirectoryError`: an operation expected a file but received a directory;
- `NotADirectoryError`: some path component expected as a directory is not one;
- `PermissionError`: the operating system denied the operation;
- broader `OSError`: other operating-system or filesystem conditions.

Catch only a failure the caller can meaningfully handle:

PYTHON
```
def read_optional_note(path):
    """Return optional note text, or an empty string when it is absent."""
    try:
        return Path(path).read_text(encoding="utf-8")
    except FileNotFoundError:
        return ""


print(repr(read_optional_note(workspace / "optional.txt")))
```

This policy is valid only because the docstring declares absence normal. A required configuration should instead let the failure propagate or add useful context. Do not catch `Exception` and report “file missing”; decoding failures, permissions, and programming bugs require different responses.

8. Build the daily field-note summary

Complete a clean, self-contained lab:

1. Create a temporary workspace with `incoming/north`, `incoming/south`, and `output` directories.
2. Write at least two `sightings.txt` inputs and one unrelated `.txt` file.
3. Implement `discover_notes(root)` to return only station notes in sorted order.

4 Implement `summarize_note(path)` to stream records and return one station total with source-and-line context for invalid counts.

5 Build the complete report in memory, end it with one newline, then write it to `output/daily-summary.txt`.

6 Run the pipeline twice and assert that discovery never includes the output and that the report text is identical.

7 Change one count to `"many"`. Confirm the exception identifies its file and line while the previous report remains available because calculation occurs before writing.

Start from these contracts:

```python
from pathlib import Path


def discover_notes(root):
    """Return station sightings files beneath root in deterministic order."""
    root = Path(root)
    return sorted(root.glob("*/sightings.txt"))


def summarize_note(path):
    """Return (station_name, total_count) for one sightings file."""
    path = Path(path)
    total = 0
    with path.open(encoding="utf-8") as handle:
        for line_number, raw_line in enumerate(handle, start=1):
            animal, raw_count = raw_line.rstrip("\n").split(",")
            if not animal:
                raise ValueError(f"{path}:{line_number}: empty animal name")
            try:
                total += int(raw_count)
            except ValueError as error:
                raise ValueError(
                    f"{path}:{line_number}: invalid count {raw_count!r}"
                ) from error
    return path.parent.name, total
```

Compare one complete integration after finishing your own

```python
from pathlib import Path
from tempfile import TemporaryDirectory

with TemporaryDirectory() as temporary_name:
    root = Path(temporary_name)
    incoming = root / "incoming"
    output = root / "output"
    (incoming / "north").mkdir(parents=True)
    (incoming / "south").mkdir()
    output.mkdir()

    (incoming / "north" / "sightings.txt").write_text(
        "otter,3\nheron,2\n", encoding="utf-8"
    )
    (incoming / "south" / "sightings.txt").write_text(
        "fox,1\notter,2\n", encoding="utf-8"
    )
    (incoming / "ignore.txt").write_text("not an input\n", encoding="utf-8")

    paths = discover_notes(incoming)
    rows = [summarize_note(path) for path in paths]
```

PYTHON (CONTINUED)

```
    report = "\n".join(f"{station}: {total}" for station, total in rows) + "\n"
    report_path = output / "daily-summary.txt"
    report_path.write_text(report, encoding="utf-8", newline="\n")

    assert [path.parent.name for path in paths] == ["north", "south"]
    assert report == "north: 5\nsouth: 3\n"
    assert report_path.read_text(encoding="utf-8") == report
    assert discover_notes(incoming) == paths
```

The narrow discovery rule makes generated output irrelevant. The report is fully constructed before `write_text` opens the destination. The next lessons will replace the simple record grammar with robust external formats and add a verified replacement step.

Checkpoint: choose the writing contract

Prevent surprising writes

Match each requirement to a deliberate writing policy.

1. When can opening an existing text file with mode `w` erase its previous contents?

- **A.** Only after the handle closes
- **B.** Only after `read()`
- **C.** As the file is opened, before later processing necessarily succeeds
- **D.** Never; `w` always appends

2. Which mode expresses “create this destination, but fail if it already exists”?

- **A.** `x`
- **B.** `a`
- **C.** `r`
- **D.** `w`

3. Why is `except Exception: return ""` a poor implementation for an optional file?

- **A.** Because optional files must be binary
- **B.** Because empty strings are invalid Python
- **C.** Because `Path` cannot raise errors
- **D.** Because it also hides permissions, decoding failures, and programming defects

Write your choices before checking the answer key at the end of this book.

9. Key points for reliable file work

- A `Path` describes a location. Resolve relative paths from a deliberate root, not an unspoken assumption about the working directory.
- Check whether a path is a file or directory when the distinction matters, but still handle operation-time failures appropriately.
- Select intended inputs and sort them when processing order affects output.

- A `with` statement closes a file handle on normal and exceptional paths; it does not hide the exception.
- Read small files whole for clarity and stream large or record-oriented files when memory and context matter.
- Modes `r`, `w`, `a`, and `x` express different contracts. In particular, `w` truncates and `write()` adds no newline automatically.
- Compute and validate complete output before opening a destination for replacement whenever practical.

10. References and next step

- Python documentation: `pathlib` — object-oriented filesystem paths
- Python documentation: `io` — streams and text I/O
- Python tutorial: reading and writing files

Next, keep the same path and resource discipline while investigating what turns stored bytes into characters—and why the wrong encoding or newline assumption can damage otherwise correct text.

Keep Text Intact Across Encodings and Newlines

Online lesson and runnable notebook

Distinguish characters from bytes, diagnose decoding failures, normalize comparisons carefully, and preserve multilingual text and line boundaries.

- **Level:** Python Foundations
- **Estimated time:** 4–5 hours
- **You will learn:** Explain the character–encoding–byte boundary, encode and decode explicitly, diagnose failures without hiding damage, compare normalized Unicode safely, control newline behavior, and replace a text artifact only after verification.
- **Practice in:** Google Colab, JupyterLab, or a local editor using temporary multilingual fixtures

A subtitle team exchanges lines such as `"Café déjà vu"`, `"Olá, Lua"`, and `"東京へようこそ"`. One machine displays them correctly. Another shows replacement symbols. A third reads every line but produces extra blank lines after export. The words did not change; assumptions at the text boundary did.

Use these questions throughout the lesson:

- Do I currently have Python characters (`str`) or encoded byte values (`bytes`)?
- Which encoding contract produced or should interpret those bytes?
- Does an error policy stop, preserve evidence, or silently discard information?
- Are two visually identical strings represented by the same code points?
- Which newline sequence is stored, and which line endings should Python expose?
- Can the previous good output survive a failed replacement attempt?

1. Text and bytes are different kinds of value

Python's `str` stores Unicode text. It is a sequence of characters as Python represents them, not a sequence of UTF-8 bytes. `bytes` stores integers from 0 to 255. An **encoding** defines how text maps to bytes; decoding applies that contract in reverse.

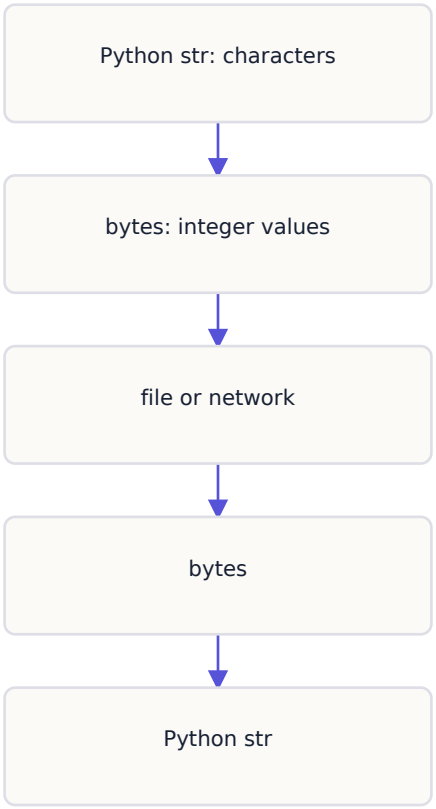
PYTHON

```
message = "Café ☕"
encoded = message.encode("utf-8")
decoded = encoded.decode("utf-8")

print(message)
print(encoded)
print("characters:", len(message))
print("bytes:", len(encoded))
print("round trip:", decoded == message)
```

The accented letter and coffee symbol need more than one UTF-8 byte, so byte length can exceed character length. The byte representation may display printable ASCII directly and other values as hexadecimal escapes. That display is evidence about bytes, not corrupted text.

Encoding and decoding cross an explicit boundary. The same codec must be part of both sides' contract for a reliable round trip.



Python str: characters → bytes: integer values: encode with UTF-8

bytes → Python str: decode with UTF-8

A file does not carry a universal promise that it is UTF-8. The producer, protocol, format, or explicit metadata must establish the encoding.

Python prevents accidental mixing:

```python
text = "Lua"
data = b"Lua"

try:
    print(text + data)
except TypeError as error:
    print(type(error).__name__, error)
```

Decide whether the operation is a text operation or a byte operation, then convert exactly at the documented boundary.

A **code point** is the numbered entry assigned to a Unicode character. `ord` shows the number and `unicodedata.name` supplies a descriptive name when one is defined:

```python
import unicodedata

for character in ["A", "é", "月", "🚀"]:
```

PYTHON (CONTINUED)

```
    print(
        repr(character),
        f"U+{ord(character):04X}",
        unicodedata.name(character),
        character.encode("utf-8").hex(" "),
    )
```

This comparison makes three measurements distinct: one displayed character, one numbered code point, and one or more UTF-8 bytes. Human-perceived symbols can be more complicated still: a flag, family emoji, or accented grapheme may contain several code points. Do not promise that `len(text)` counts what a user would call visible characters.

2. An explicit encoding makes the file contract portable

The default text encoding can vary across platforms, locales, and Python configuration. State the encoding when reading or writing durable data:

PYTHON

```
from pathlib import Path
from tempfile import TemporaryDirectory

with TemporaryDirectory() as temporary_name:
    workspace = Path(temporary_name)
    subtitle_path = workspace / "opening.txt"
    subtitle_path.write_text("Olá, Lua!\n東京へようこそ\n", encoding="utf-8")

    restored = subtitle_path.read_text(encoding="utf-8")
    print(restored)
    assert restored == "Olá, Lua!\n東京へようこそ\n"
```

Here, both operations promise UTF-8. Writing first makes the notebook example self-contained; reading proves the round trip rather than depending on an unknown file.

What happens if the declared decoding contract is wrong?

PYTHON

```
text = "Café"
utf8_data = text.encode("utf-8")

try:
    utf8_data.decode("ascii")
except UnicodeDecodeError as error:
    print("codec:", error.encoding)
    print("start:", error.start)
    print("end:", error.end)
    print("reason:", error.reason)
    print("offending bytes:", error.object[error.start:error.end])
```

`UnicodeDecodeError` identifies the attempted codec and byte position. Preserve that evidence. Repeatedly trying random encodings until one produces characters can yield plausible-looking but false text.

The absence of an exception does not prove the encoding guess was correct. These Latin-1 bytes decode under a different single-byte codec into the wrong characters, then round-trip perfectly under that wrong assumption:

```python
utf8_data = "Café".encode("utf-8")
wrong_text = utf8_data.decode("latin-1")
wrong_round_trip = wrong_text.encode("latin-1")

print(wrong_text)
assert wrong_round_trip == utf8_data
```

The display becomes the familiar mojibake `CafÃ©`, yet the bytes round-trip. A round-trip verifies internal consistency with a chosen codec; it cannot discover the producer's contract. Use metadata, a protocol, or knowledge of the source. Re-encoding mojibake repeatedly can make recovery harder, so retain the original bytes while investigating.

Encoding can fail too:

```python
try:
    "東京".encode("ascii")
except UnicodeEncodeError as error:
    print(type(error).__name__, error)
```

ASCII has no representation for those characters. The solution is normally to choose the correct output encoding—often UTF-8—not delete the characters.

3. Error policies trade fidelity for continuation

The default `errors="strict"` raises when conversion cannot satisfy the codec. That is the safest default because it stops at the boundary instead of hiding loss.

```python
invalid_utf8 = b"signal:\xff:end"

for policy in ["replace", "backslashreplace", "ignore"]:
    result = invalid_utf8.decode("utf-8", errors=policy)
    print(policy, repr(result))
```

Typical results illustrate three policies:

- `replace` inserts `�`, visibly marking damage;
- `backslashreplace` exposes the offending byte as text such as `\\xff`;
- `ignore` drops information and can join text that was never adjacent.

Continuation is not correctness

Use a non-strict policy only when the application contract says how data loss is recorded and reviewed. `errors="ignore"` should not be a reflexive "fix" for an unknown input. Keep the original bytes so another tool or corrected contract can recover them later. :::

Checkpoint: protect the character–byte contract

Know which side of the boundary you hold

Choose the statement that preserves the encoding contract.

1. Why can `len(text.encode("utf-8"))` exceed `len(text)`?

- **A.** Some characters require multiple UTF-8 bytes

- **B.** Encoding adds a Python list around every character
- **C.** All files duplicate each character
- **D.** `len` counts words for strings

2. What is the best first response to a `UnicodeDecodeError`?

- **A.** Delete the file
- **B.** Decode with `errors="ignore"` and forget the loss
- **C.** Record the attempted codec and failing byte position, then verify the source contract
- **D.** Rename the suffix to `.utf8`

3. Why specify `encoding="utf-8"` for a durable text file?

- **A.** It makes every file smaller
- **B.** It avoids relying on a platform-dependent default
- **C.** It converts CSV into JSON
- **D.** It closes the file twice

Write your choices before checking the answer key at the end of this book.

4. Byte-order marks require a deliberate policy

Some UTF-8 text begins with the byte sequence `EF BB BF`, commonly called a UTF-8 byte-order mark (BOM). It is unnecessary for UTF-8 byte order, but tools sometimes emit it as a signature. Decoding with plain `utf-8` exposes a leading Unicode character `\ufeff`; `utf-8-sig` consumes it when present.

PYTHON

```
with_bom = b"\xef\xbb\xbfheadline\n"

print(repr(with_bom.decode("utf-8")))
print(repr(with_bom.decode("utf-8-sig")))
```

This matters when the first CSV header unexpectedly becomes `"\ufeffname"` instead of `"name"`. If your input contract permits a UTF-8 BOM, opening with `encoding="utf-8-sig"` handles files with or without it. Do not strip `\ufeff` from arbitrary positions; the policy belongs at the start of a documented text stream.

5. Visually identical text can use different code points

Unicode sometimes represents the same visible text in more than one way. `é` can be one precomposed code point or an `e` followed by a combining accent.

PYTHON

```
import unicodedata

composed = "Café"
decomposed = "Cafe\u0301"

print(composed == decomposed)
print([f"U+{ord(character):04X}" for character in composed])
print([f"U+{ord(character):04X}" for character in decomposed])

normalized_left = unicodedata.normalize("NFC", composed)
```

PYTHON (CONTINUED)

```
normalized_right = unicodedata.normalize("NFC", decomposed)
print(normalized_left == normalized_right)
```

Normalization form NFC is useful when an application wants canonically equivalent text to compare equal. NFD decomposes compatible sequences instead. Other forms, NFKC and NFKD, perform compatibility transformations that can change distinctions; do not choose them without a domain reason.

Preserve original display text when evidence matters:

PYTHON

```
record = {
    "display_name": decomposed,
    "comparison_name": unicodedata.normalize("NFC", decomposed),
}

assert record["display_name"] == decomposed
assert record["comparison_name"] == unicodedata.normalize("NFC", composed)
```

Normalization is not case folding, translation, spell checking, or security validation. It answers one narrower representation question.

6. Newlines have stored and in-memory forms

Common line terminators include:

- line feed: `\n`, conventional on Unix-like systems;
- carriage return plus line feed: `\r\n`, conventional on Windows;
- carriage return: `\r`, present in some older data.

`repr` makes invisible separators visible:

PYTHON

```
mixed = "north\r\nsouth\nwest\r"
print(repr(mixed))
print(mixed.splitlines())
```

`splitlines()` understands several line boundaries and removes them by default. Use `split("\n")` only when the contract specifically requires a line-feed separator and you want its exact empty-field behavior.

When a text file is opened with `newline=None` (the default), Python applies universal newline translation on reading: `\r`, `\r\n`, and `\n` become `\n` in the returned text. The handle's `.newlines` attribute reports terminators it has observed:

PYTHON

```
from pathlib import Path
from tempfile import TemporaryDirectory

with TemporaryDirectory() as temporary_name:
    path = Path(temporary_name) / "mixed-lines.txt"
    path.write_bytes(b"alpha\r\nbeta\ngamma\r")

    with path.open(encoding="utf-8", newline=None) as handle:
        translated = handle.read()
        observed = handle.newlines
```

PYTHON (CONTINUED)

```
    with path.open(encoding="utf-8", newline="") as handle:
        preserved = handle.read()

    print("translated:", repr(translated))
    print("observed:", observed)
    print("preserved:", repr(preserved))
```

With `newline=""`, line endings are returned without translation. This is the setting required by Python's CSV module because the CSV parser owns record newline handling. For ordinary application text, translation is often useful.

Be precise when removing terminators. `line.rstrip()` removes *all* trailing whitespace, including spaces and tabs that may be meaningful. `rstrip("\r\n")` removes any run of those characters, while `splitlines()` expresses the intent to divide lines. Choose according to the data contract.

Compare a subtitle whose indentation belongs to its display:

PYTHON

```
indented_subtitle = "    whisper from the left    \r\n"

print("strip:", repr(indented_subtitle.strip()))
print("newline only:", repr(indented_subtitle.rstrip("\r\n")))
```

The unqualified `strip()` removes both leading and trailing spaces as well as the terminator. That may be correct for an identifier field and destructive for preformatted text. The operation name alone does not define the policy; the field contract does.

A short matrix makes newline choices easier to review:

Operation	Preserves terminator?	Preserves surrounding spaces?	Typical use
`splitlines()`	no	yes	divide ordinary text into logical lines
`splitlines(keepends=True)`	yes	yes	inspect or reproduce exact boundaries
`rstrip("\r\n")`	no trailing CR/LF run	yes	remove a known record terminator
`strip()`	no	no	normalize a field only when all edge whitespace is insignificant

A final newline is also a policy. Many line-oriented tools expect one. Build it deliberately:

PYTHON

```
subtitle_lines = ["Olá, Lua!", "東京へようこそ"]
subtitle_text = "\n".join(subtitle_lines) + "\n"
assert subtitle_text.endswith("\n")
```

Checkpoint: compare and split text deliberately

Preserve meaning across representations

Choose the narrow policy that matches the stated need.

1. A CSV producer may include a UTF-8 BOM. Which decoding contract handles it at the start of the stream?

- **A.** ASCII with ignore
- **B.** UTF-16 for every file
- **C.** Strip every unusual character after parsing
- **D.** `encoding="utf-8-sig"`

2. What does NFC normalization help an application compare?

- **A.** Files with different permissions
- **B.** Canonically equivalent Unicode sequences such as composed and decomposed accents
- **C.** All misspelled words
- **D.** Binary images and text

3. Which open setting lets a format parser such as `csv` receive newline characters without text translation?

- **A.** `newline=""`
- **B.** `errors="ignore"`
- **C.** `mode="x"`
- **D.** Omit both encoding and newline forever

Write your choices before checking the answer key at the end of this book.

7. Replace an output only after creating and checking it

Writing directly to a known-good destination can leave it empty or partial if generation fails. A safer local workflow writes a sibling temporary artifact, reads it back under the promised contract, and only then replaces the final path.

PYTHON

```
from pathlib import Path


def replace_verified_text(path, text):
    """Write verified UTF-8 text beside path, then replace path."""
    path = Path(path)
    temporary_path = path.with_name(path.name + ".tmp")

    try:
        temporary_path.write_text(text, encoding="utf-8", newline="\n")
        restored = temporary_path.read_text(encoding="utf-8")
        if restored != text:
            raise OSError("temporary text did not round-trip")
        temporary_path.replace(path)
    except Exception:
        temporary_path.unlink(missing_ok=True)
        raise

    return path
```

`Path.replace()` requests a filesystem replacement. Keeping the temporary file in the same directory avoids an unnecessary cross-filesystem move. This pattern reduces the window in which the

destination is incomplete, but it is **not a universal durability or transaction guarantee**. Filesystem, operating-system, power-loss, permissions, locking, and directory-sync behavior matter in high-stakes systems. State the scope of the promise you have actually tested.

Exercise both paths:

```python
from tempfile import TemporaryDirectory

with TemporaryDirectory() as temporary_name:
    output_path = Path(temporary_name) / "subtitles.txt"
    output_path.write_text("previous good version\n", encoding="utf-8")

    new_text = "Olá, Lua!\n東京へようこそ\n"
    replace_verified_text(output_path, new_text)

    assert output_path.read_text(encoding="utf-8") == new_text
    assert not (output_path.parent / "subtitles.txt.tmp").exists()
```

For JSON, Lesson 3 will verify that the temporary text parses back to the expected Python structure, a stronger check than text equality alone.

8. Restore the multilingual subtitle reel

Build a lab with four supplied byte payloads:

```python
subtitle_payloads = {
    "opening-utf8.txt": "Olá, Lua!\n".encode("utf-8"),
    "tokyo-bom.txt": b"\xef\xbb\xbf" + "東京へようこそ\r\n".encode("utf-8"),
    "cafe-decomposed.txt": "Cafe\u0301\n".encode("utf-8"),
    "damaged.txt": b"signal:\xff:end\n",
}
```

Your tasks:

1. Write the payloads as bytes in a temporary `incoming` directory.
2. Decode the first three with a documented UTF-8/BOM policy and preserve their original display strings.
3. Store NFC comparison forms separately and prove the decomposed café matches `"Café"` after normalization.
4. Use `repr` to record the translated and preserved line endings of the BOM file.
5. Attempt strict decoding of `damaged.txt`; capture codec, position, and offending bytes in a rejection record without deleting the original.
6. Join accepted display lines with `\n`, add one final newline, and call `replace_verified_text`.
7. Rerun from a clean workspace and compare exact output text.

Inspect one recovery loop after completing your investigation

```python
import unicodedata
from pathlib import Path
from tempfile import TemporaryDirectory

with TemporaryDirectory() as temporary_name:
```

PYTHON (CONTINUED)

```
    root = Path(temporary_name)
    incoming = root / "incoming"
    incoming.mkdir()
    for name, payload in subtitle_payloads.items():
        (incoming / name).write_bytes(payload)

    accepted = []
    rejected = []
    for path in sorted(incoming.glob("*.txt")):
        try:
            display_text = path.read_text(encoding="utf-8-sig")
        except UnicodeDecodeError as error:
            rejected.append(
                {
                    "source": path.name,
                    "encoding": error.encoding,
                    "position": error.start,
                    "bytes": error.object[error.start:error.end].hex(),
                }
            )
            continue

        accepted.append(
            {
                "source": path.name,
                "display": display_text,
                "comparison": unicodedata.normalize("NFC", display_text),
            }
        )

    cafe = next(item for item in accepted if item["source"] == "cafe-decomposed.txt")
    assert cafe["comparison"].strip() == "Café"
    assert rejected[0]["source"] == "damaged.txt"

    output_text = "".join(item["display"].replace("\r\n", "\n") for item in accepted)
    if output_text and not output_text.endswith("\n"):
        output_text += "\n"
    output_path = replace_verified_text(root / "restored.txt", output_text)
    assert output_path.read_text(encoding="utf-8") == output_text
    saved_bytes = output_path.read_bytes()
    assert saved_bytes.decode("utf-8") == output_text
    assert saved_bytes.endswith(b"\n")
```

The original bytes remain available. Accepted records preserve display text and store a separate normalized comparison value. The rejection explains why one payload could not cross the declared boundary. Inspecting the saved bytes closes the whole contract: Python text was encoded as UTF-8, the result decodes back to the intended text, and the byte artifact retains the chosen final newline.

Checkpoint: publish text without hiding damage

Choose evidence-preserving output behavior

Select the policy that matches a trustworthy text pipeline.

1. Why is strict decoding the safest default for an archive import?

- **A.** It silently repairs every codec
- **B.** It converts bytes into CSV rows
- **C.** It removes all Unicode characters
- **D.** It stops at undecodable data instead of concealing information loss

2. What should happen before a temporary artifact replaces the known-good text output?

- **A.** Read it back under the promised contract and verify the result
- **B.** Delete both the old and temporary files
- **C.** Change the source encoding randomly
- **D.** Ignore every write exception

3. Why should the lesson avoid calling `Path.replace()` a universal transaction guarantee?

- **A.** Because paths cannot be replaced
- **B.** Because only JSON can be durable
- **C.** Because temporary files are always remote
- **D.** Because filesystem, OS, power-loss, and durability semantics still affect the promise

Write your choices before checking the answer key at the end of this book.

9. Key points for text integrity

- `str` contains Unicode text; `bytes` contains integer byte values. Encoding and decoding cross between them under an explicit codec contract.
- A decoding exception provides useful byte-position evidence. Do not replace it with silent loss unless the application explicitly records that trade-off.
- `utf-8-sig` is useful when a producer may place a UTF-8 BOM at the beginning of a text stream.
- Unicode normalization can align canonically equivalent comparison forms while original display text remains preserved.
- Newline translation, splitting, stripping, and final-newline choices express different policies; choose them deliberately.
- Create and verify a sibling temporary artifact before replacement when a previous good local output should survive generation failures.

10. References and next step

- Python Unicode HOWTO
- Python documentation: codecs and encoding error handlers
- Python documentation: text I/O and newline behavior
- Python documentation: Unicode normalization

Next, apply these encoding and newline contracts to structured records. CSV, JSON, and YAML each preserve a different set of information and require their own parser and writer rather than ad hoc string operations.

Move Records Between CSV, JSON, YAML, and Python

Online lesson and runnable notebook

Choose a structured-text format, use its reader and writer correctly, convert types explicitly, and validate parsed values before trusting them.

- **Level:** Python Foundations
- **Estimated time:** 5–6 hours
- **You will learn:** Choose between CSV, JSON, and YAML; handle quoted CSV records and headers; distinguish stream and string APIs; serialize Unicode predictably; use safe YAML loading; and validate structure and types after parsing.
- **Practice in:** Google Colab, JupyterLab, or a local editor with temporary festival data

A neighborhood festival receives booth registrations from a spreadsheet, saves a processed snapshot for another program, and keeps a small human-edited configuration. Those three jobs look like “store some records,” yet they have different shapes and different readers:

- CSV is naturally tabular and exchanges well with spreadsheets;
- JSON represents nested combinations of objects, arrays, strings, numbers, booleans, and null; and
- YAML is convenient for human-edited configuration but has a larger grammar and requires a third-party Python library here.

The format parser answers “Is this valid CSV/JSON/YAML syntax?” Your application must still answer “Is this the kind of festival record we accept?” Keep those two questions separate.

1. Choose a format from the data and its consumers

Need	CSV	JSON	YAML
rows with a shared set of columns	strong fit	possible, more punctuation	possible
deeply nested data	awkward	strong fit	strong fit
spreadsheet exchange	common	less convenient	uncommon
human-edited comments	no standard comment facility	no comments	supported
Python standard library	`csv`	`json`	no; this course uses PyYAML
types after parsing	fields are strings (or `None` in special cases)	defined JSON-to-Python mapping	loader-dependent scalar typing

A filename suffix communicates a convention; it does not validate the contents. Use the format’s parser, then validate the resulting Python values.

Write the interchange contract down

Record the encoding, newline policy, fields/keys, permitted types, missing-value rules, and version expectations. “It’s a CSV” is not enough: CSV has dialect, header, quoting, and application-schema choices. :::

2. CSV quoting defeats manual comma splitting

Consider this valid CSV text:

PYTHON
```
csv_text = '''booth_id,name,description,capacity
B-101,Café Aurora,"Coffee, tea, and cakes",40
B-102,月の屋台,"Games on two
levels",25
'''

for physical_line in csv_text.splitlines():
    print(physical_line.split(","))
```

The first description contains a quoted comma. The second contains a quoted newline, so one logical record spans two physical lines. Splitting strings on commas or newlines cannot implement the CSV grammar correctly.

Use `csv.reader` for positional rows or `csv.DictReader` when the first row names fields:

PYTHON
```
import csv
from io import StringIO

with StringIO(csv_text, newline="") as handle:
    reader = csv.DictReader(handle)
    rows = list(reader)
    last_line_number = reader.line_num

print(rows[0])
print(repr(rows[1]["description"]))
print("last physical line consumed:", last_line_number)
```

Every ordinary CSV field is text. `"40"` is not automatically the integer `40`. Convert under an application rule and add context to failure:

`csv.reader` is useful when position is the public contract or when a file has no header. Dialect options belong to the reader or writer rather than string replacement:

PYTHON
```
semicolon_text = 'B-201;"Tea; coffee";30\n'

with StringIO(semicolon_text, newline="") as handle:
    reader = csv.reader(
        handle,
        delimiter=";",
        quotechar='"',
        quoting=csv.QUOTE_MINIMAL,
    )
    positional_row = next(reader)

assert positional_row == ["B-201", "Tea; coffee", "30"]
```

Only choose a non-default delimiter because the producer’s dialect specifies it. Python cannot infer every CSV variation reliably from one short sample.

```python
def parse_capacity(raw_value, source, record_number):
    """Return a positive integer capacity with source context."""
    try:
        capacity = int(raw_value)
    except (TypeError, ValueError) as error:
        raise ValueError(
            f"{source}: record {record_number}: invalid capacity {raw_value!r}"
        ) from error
    if capacity <= 0:
        raise ValueError(
            f"{source}: record {record_number}: capacity must be positive"
        )
    return capacity


print(parse_capacity(rows[0]["capacity"], "booths.csv", 1))
```

`reader.line_num` counts physical lines read, which is not always the same as a logical record number when quoted fields contain newlines. Track a logical record number with `enumerate(reader, start=1)` and include both kinds of context when useful.

3. Open CSV files with an empty newline setting

The `csv` documentation requires files to be opened with `newline=""`. That lets the CSV module, rather than the text layer, own CSV newline handling. Continue to state the encoding.

```python
from pathlib import Path
from tempfile import TemporaryDirectory

format_temporary_directory = TemporaryDirectory()
workspace = Path(format_temporary_directory.name)
csv_path = workspace / "booths.csv"
csv_path.write_text(csv_text, encoding="utf-8", newline="")

with csv_path.open(encoding="utf-8", newline="") as handle:
    reader = csv.DictReader(handle)
    print("headers:", reader.fieldnames)
    file_rows = list(reader)

assert len(file_rows) == 2
```

Validate headers before trusting rows:

```python
EXPECTED_HEADERS = ["booth_id", "name", "description", "capacity"]


def require_csv_headers(fieldnames, source):
    """Raise when CSV headers do not exactly match the versioned contract."""
    if fieldnames != EXPECTED_HEADERS:
        raise ValueError(
            f"{source}: expected headers {EXPECTED_HEADERS!r}; got {fieldnames!r}"
        )
```

An exact-order policy is easy to explain but may be stricter than some systems need. Another application might allow any order while rejecting missing and extra names separately. Decide deliberately. `DictReader` uses `None` as a key for surplus fields and `restval` for missing ones; check rather than assuming a short or long row is harmless.

Make those shapes visible with a controlled input:

```python
uneven_text = "id,name\n1,Ana,EXTRA\n2\n"
with StringIO(uneven_text, newline="") as handle:
    uneven_reader = csv.DictReader(handle, restkey="_extra", restval="<missing>")
    uneven_rows = list(uneven_reader)

assert uneven_rows[0]["_extra"] == ["EXTRA"]
assert uneven_rows[1]["name"] == "<missing>"
```

These sentinels expose malformed row shape. They do not decide whether extra or missing fields are permitted; validation owns that decision.

Write through `csv.writer` or `csv.DictWriter` so quoting is handled correctly:

```python
output_path = workspace / "accepted-booths.csv"
output_rows = [
    {
        "booth_id": "B-101",
        "name": "Café Aurora",
        "description": "Coffee, tea, and cakes",
        "capacity": 40,
    },
    {
        "booth_id": "B-102",
        "name": "月の屋台",
        "description": "Games on two\nlevels",
        "capacity": 25,
    },
]

with output_path.open("w", encoding="utf-8", newline="") as handle:
    writer = csv.DictWriter(handle, fieldnames=EXPECTED_HEADERS)
    writer.writeheader()
    writer.writerows(output_rows)

with output_path.open(encoding="utf-8", newline="") as handle:
    restored_rows = list(csv.DictReader(handle))

assert restored_rows[0]["description"] == "Coffee, tea, and cakes"
assert restored_rows[0]["capacity"] == "40"
```

The integer became CSV text and comes back as a string. That is expected format behavior, not a failed round trip. Define equality in terms of the interchange contract, not Python object identity.

`csv.writer` writes sequences, while `DictWriter` selects named fields in its declared order. Values are converted to text; notably, the module writes `None` as an empty field, which cannot be reversed without a separate missing-value contract:

```python
none_stream = StringIO(newline="")
csv.writer(none_stream).writerow(["B-999", None, ""])
print(repr(none_stream.getvalue()))
```

If `None` and an intentionally empty string mean different things, choose an explicit sentinel or schema instead of assuming CSV preserves the distinction.

Checkpoint: read tables as CSV, not punctuation

Preserve complete CSV records

Choose the behavior required by a robust CSV boundary.

1. Why does `line.split(",")` fail as a general CSV parser?

- **A.** CSV never contains commas
- **B.** Quoted fields can contain commas, quotes, and newlines that belong to one field
- **C.** Python strings cannot be split
- **D.** CSV is always binary

2. How should a text file be opened for Python's CSV reader or writer?

- **A.** Without an encoding and with `errors="ignore"`
- **B.** In binary append mode
- **C.** With `newline=""` and an explicit encoding
- **D.** By reading each physical line into `split`

3. After `DictReader` reads a field containing `40`, what should application code normally expect?

- **A.** An automatically validated positive integer
- **B.** A YAML value
- **C.** A file handle
- **D.** The string `"40"`, which must be converted under a validation rule

Write your choices before checking the answer key at the end of this book.

4. JSON maps a defined set of values

JSON supports objects, arrays, strings, numbers, booleans, and null. Python's `json` module maps them approximately as follows:

JSON	Python after loading
object	`dict`
array	`list`
string	`str`
integer number	`int`
non-integer number	`float`
`true` / `false`	`True` / `False`
`null`	`None`

`loads` and `dumps` work with strings. `load` and `dump` work with open file-like objects:

```python
import json

festival = {
    "name": "Festival da Lua",
    "open": True,
    "visitor_limit": 500,
    "booths": ["Café Aurora", "月の屋台"],
    "closed_at": None,
}

json_text = json.dumps(
    festival,
    ensure_ascii=False,
    indent=2,
    sort_keys=True,
)
restored = json.loads(json_text)

print(json_text)
assert restored == festival
```

`ensure_ascii=False` keeps non-ASCII characters readable in the Unicode output string. Encoding remains a separate file operation; write that string as UTF-8. `indent=2` makes a human-readable artifact, while `sort_keys=True` makes object key order deterministic for comparisons. Neither option validates the data.

Some Python values have no direct JSON representation:

```python
from pathlib import Path

unsupported = {"path": Path("booths.csv"), "tags": {"food", "games"}}

try:
    json.dumps(unsupported)
except TypeError as error:
    print(type(error).__name__, error)
```

Convert intentionally at your application's boundary—for example, a path to a string and a set to a sorted list. Avoid a catch-all `default=str` when it would turn unsupported or accidental objects into ambiguous text.

Even supported-looking containers can change representation:

```python
round_trip_cases = {
    "tuple": ("north", "south"),
    "integer_key": {7: "lucky"},
    "nested": {"values": [True, None, 3.5]},
}

for name, value in round_trip_cases.items():
    restored_value = json.loads(json.dumps(value))
    print(name, repr(value), "->", repr(restored_value))
```

A tuple returns as a list, and an integer object key returns as the string `"7"`, because JSON arrays do not record Python tuple identity and JSON object names are strings. Dates and times also need an explicit representation such as an ISO 8601 string plus a documented timezone policy. Test semantic round trips for the data model your application actually promises.

5. JSON parsing failures include location evidence

Malformed JSON raises `JSONDecodeError`, a subclass of `ValueError`:

```python
broken_json = '{"name": "Aurora", "capacity": 40,}'

try:
    json.loads(broken_json)
except json.JSONDecodeError as error:
    print("message:", error.msg)
    print("line:", error.lineno)
    print("column:", error.colno)
    print("character:", error.pos)
```

The parser error says where JSON grammar failed. If parsing succeeds but `capacity` is negative or `booths` is a string instead of a list, that is an application validation failure and should have a different message.

Use one `dump` per complete JSON document. JSON is not a framed protocol, so calling `dump` repeatedly on one stream normally concatenates invalid documents:

```python
from io import StringIO

stream = StringIO()
json.dump({"id": 1}, stream)
json.dump({"id": 2}, stream)
print(stream.getvalue())
```

If you need a sequence, write one JSON array, adopt a specified format such as newline-delimited JSON, or use another framing protocol. State which contract readers expect.

Untrusted JSON can consume substantial CPU and memory through huge inputs or deep structures. Limit input size at the surrounding system boundary when that risk matters; successful JSON grammar parsing is not a resource-safety proof.

Write a complete deterministic document with a final newline, then parse it back before publishing:

```python
archive_path = workspace / "festival.json"
archive_text = json.dumps(
    festival,
    ensure_ascii=False,
    indent=2,
    sort_keys=True,
) + "\n"
archive_path.write_text(archive_text, encoding="utf-8", newline="\n")

with archive_path.open(encoding="utf-8") as handle:
    restored_from_file = json.load(handle)

assert restored_from_file == festival
```

Checkpoint: make JSON output predictable

Separate JSON syntax from application rules

Choose the narrow claim supported by the JSON operation.

1. Which statement correctly distinguishes `json.load` and `json.loads`?

- **A.** Both only read CSV
- **B.** `load` validates business rules while `loads` does not
- **C.** `loads` writes bytes to a path
- **D.** `load` reads a file-like object; `loads` parses a string or compatible byte input

2. What does `ensure_ascii=False` do when dumping JSON to a Python string?

- **A.** It preserves non-ASCII characters visibly instead of escaping every one
- **B.** It chooses a filesystem path
- **C.** It validates every key
- **D.** It turns JSON into YAML

3. Why not call `json.dump` repeatedly to append independent objects to one ordinary JSON document?

- **A.** Objects cannot contain numbers
- **B.** JSON is not framed, so adjacent dumped values do not form one normal document
- **C.** `dump` only accepts CSV rows
- **D.** Files can contain only one character

Write your choices before checking the answer key at the end of this book.

6. YAML is a third-party, human-edited boundary

This course uses PyYAML 6 or newer as a direct project dependency. In a fresh standalone notebook, the following hidden setup cell installs it only when it is absent:

```python
import importlib.util
import subprocess
import sys

if importlib.util.find_spec("yaml") is None:
    subprocess.run(
        [sys.executable, "-m", "pip", "install", "PyYAML>=6"],
        check=True,
    )
```

Now load the public API:

```python
import yaml
```

For untrusted or ordinary configuration, use `safe_load`, not a loader capable of constructing arbitrary Python objects:

```python
yaml_text = (
    "festival_name: Festival da Lua\n"
    "open: true\n"
    "visitor_limit: 500\n"
    "theme:\n"
    "  primary_color: indigo\n"
```

PYTHON (CONTINUED)

```
    "  languages:\n"
    "    - pt\n"
    "    - ja\n"
)

configuration = yaml.safe_load(yaml_text)
print(configuration)
assert configuration["open"] is True
```

Safe loading restricts object construction. It does **not** prove that the result is a mapping, that required keys exist, or that values satisfy festival rules. Even an empty YAML document can parse to `None`.

PYTHON

```
def validate_configuration(value):
    """Return a small validated festival configuration."""
    if not isinstance(value, dict):
        raise ValueError("configuration must be a mapping")

    required = {"festival_name", "open", "visitor_limit", "theme"}
    missing = required - value.keys()
    extra = value.keys() - required
    if missing:
        raise ValueError(f"missing configuration keys: {sorted(missing)!r}")
    if extra:
        raise ValueError(f"unexpected configuration keys: {sorted(extra)!r}")
    if not isinstance(value["festival_name"], str) or not value["festival_name"].strip():
        raise ValueError("festival_name must be non-empty text")
    if type(value["open"]) is not bool:
        raise ValueError("open must be a boolean")
    if not isinstance(value["visitor_limit"], int) or isinstance(
        value["visitor_limit"], bool
    ):
        raise ValueError("visitor_limit must be an integer")
    if value["visitor_limit"] <= 0:
        raise ValueError("visitor_limit must be positive")
    if not isinstance(value["theme"], dict):
        raise ValueError("theme must be a mapping")

    return value


validated_configuration = validate_configuration(configuration)
```

The explicit `type(... ) is not bool` check matters because `bool` is a subclass of `int` in Python. Domain contracts sometimes need stricter distinctions than `isinstance(value, int)` alone.

Write with `safe_dump`:

PYTHON

```
yaml_output = yaml.safe_dump(
    validated_configuration,
    allow_unicode=True,
    sort_keys=True,
)
print(yaml_output)
assert validate_configuration(yaml.safe_load(yaml_output)) == configuration
```

`allow_unicode=True` keeps multilingual text readable. YAML formatting and scalar interpretation can vary by library and version, so compare validated values rather than requiring one hand-written spacing style unless textual form is itself the contract.

Inspect scalar typing rather than assuming every unquoted value remains text:

```python
scalar_sample = yaml.safe_load(
    "count: 7\nratio: 1.5\nactive: true\nmissing: null\nlabel: '007'\n"
)

assert scalar_sample == {
    "count": 7,
    "ratio": 1.5,
    "active": True,
    "missing": None,
    "label": "007",
}
```

Quoting `007` preserves it as a label instead of inviting numeric interpretation. PyYAML can also construct date-like scalars as date objects. Treat such behavior as part of the chosen loader/version contract, and validate types immediately after loading.

7. Compare the same record across format boundaries

Suppose one accepted booth is:

```python
comparison_record = {
    "booth_id": "B-101",
    "name": "Café Aurora",
    "capacity": 40,
    "features": ["tea", "games"],
}
```

CSV needs a policy for turning `features` into one cell or a related table. JSON and YAML can represent the list directly. YAML can carry human comments, but those comments normally do not survive a `safe_load`/`safe_dump` value round trip. JSON is broadly interoperable but deliberately has no comment syntax. No format preserves Python classes, aliases, validators, or business meaning automatically.

Use this decision record before implementation:

Question	Example answer for the festival
Who produces and consumes it?	spreadsheet staff produce booth rows; programs consume the archive
What is the natural shape?	flat rows for intake, nested values for snapshot
Must people edit it?	YAML configuration is reviewed by coordinators
Which types must survive?	capacity is validated as integer; feature order is meaningful
How is invalid data represented?	raw CSV row plus source, logical record, and reason
What makes output reproducible?	stable source order, key order, field order, encoding, and newline policy

8. Build one festival conversion pipeline

Create three files in a temporary workspace:

- `incoming/booths.csv` with a quoted comma, a quoted multiline description, one invalid capacity, and multilingual names;
- `config/festival.yml` with a positive visitor limit and theme; and
- `output/festival.json`, which does not exist yet.

Build functions with these responsibilities:

```python
def load_booths(path):
    """Return (accepted, rejected) booth records from a UTF-8 CSV file."""
    # Validate headers, enumerate logical records, and convert capacity.
    ...


def load_festival_configuration(path):
    """Return validated configuration loaded with yaml.safe_load."""
    ...


def build_festival_archive(configuration, booths):
    """Return the JSON-compatible archive value."""
    return {"configuration": configuration, "booths": booths}
```

Requirements:

1. Open CSV with `encoding="utf-8"` and `newline=""`.
2. Require the exact header contract.
3. Preserve the raw CSV row and a useful reason for each rejected record.
4. Do not catch unexpected exceptions from programming defects.
5. Load YAML safely, then validate its Python structure.
6. Dump one JSON document with visible Unicode, indentation, deterministic keys, and one final newline.
7. Parse the output back and compare it to the Python archive.
8. Rerun the pipeline and prove the serialized text is identical.

Compare the boundary order after completing your version

```python
def load_booths(path):
    accepted = []
    rejected = []
    path = Path(path)
    with path.open(encoding="utf-8", newline="") as handle:
        reader = csv.DictReader(handle)
        require_csv_headers(reader.fieldnames, path.name)
        for record_number, row in enumerate(reader, start=1):
            try:
                booth_id = row["booth_id"].strip()
                name = row["name"].strip()
                if not booth_id or not name:
                    raise ValueError("booth_id and name must be non-empty")
                capacity = parse_capacity(
                    row["capacity"], path.name, record_number
                )
            except ValueError as error:
                rejected.append(
                    {
```

PYTHON (CONTINUED)

```
                        "source": path.name,
                        "record": record_number,
                        "raw": dict(row),
                        "reason": str(error),
                    }
                )
            else:
                accepted.append(
                    {
                        "booth_id": booth_id,
                        "name": name,
                        "description": row["description"],
                        "capacity": capacity,
                    }
                )
    return accepted, rejected


def load_festival_configuration(path):
    with Path(path).open(encoding="utf-8") as handle:
        return validate_configuration(yaml.safe_load(handle))
```

Here is a complete caller using a fresh fixture and a parse-back check:

PYTHON

```
integration_root = workspace / "integration"
incoming = integration_root / "incoming"
configuration_directory = integration_root / "config"
output_directory = integration_root / "output"
for directory in [incoming, configuration_directory, output_directory]:
    directory.mkdir(parents=True, exist_ok=True)

(incoming / "booths.csv").write_text(
    (
        "booth_id,name,description,capacity\n"
        'B-101,Café Aurora,"Coffee, tea",40\n'
        'B-102,月の屋台,"Games on two\nlevels",25\n'
        "B-103,Broken Booth,Invalid capacity,many\n"
    ),
    encoding="utf-8",
    newline="",
)
(configuration_directory / "festival.yml").write_text(
    yaml_text,
    encoding="utf-8",
)

accepted_booths, rejected_booths = load_booths(incoming / "booths.csv")
festival_configuration = load_festival_configuration(
    configuration_directory / "festival.yml"
)
complete_archive = {
    **build_festival_archive(festival_configuration, accepted_booths),
    "rejected": rejected_booths,
}
complete_text = json.dumps(
    complete_archive,
    ensure_ascii=False,
    indent=2,
    sort_keys=True,
) + "\n"
complete_path = output_directory / "festival.json"
complete_path.write_text(complete_text, encoding="utf-8", newline="\n")

with complete_path.open(encoding="utf-8") as handle:
    parsed_archive = json.load(handle)
```

PYTHON (CONTINUED)

```
assert parsed_archive == complete_archive
assert len(parsed_archive["booths"]) == 2
assert len(parsed_archive["rejected"]) == 1
assert "Café Aurora" in complete_text
assert complete_text.endswith("\n")
```

The caller keeps CSV parsing, YAML configuration validation, archive assembly, and JSON verification visibly separate. Lesson 2's temporary replacement helper can guard the local text write; the challenge combines JSON parse-back verification with replacement directly.

Checkpoint: choose the complete format boundary

Parse safely, then validate

Choose what each layer can actually guarantee.

1. Which operation should load an ordinary PyYAML configuration from an untrusted or external source?

- **A.** `yaml.load` with an arbitrary object loader
- **B.** `eval` on the YAML text
- **C.** `yaml.safe_load`, followed by application validation
- **D.** Manual splitting on colons

2. A JSON document parses successfully but contains `"visitor_limit": -5`. What has been proven?

- **A.** Only that the document follows JSON syntax; the domain rule can still reject it
- **B.** That every festival accepts negative limits
- **C.** That the file is CSV
- **D.** That no further validation is needed

3. Which combination best fits spreadsheet booth rows, a nested program snapshot, and human-edited configuration?

- **A.** YAML, YAML, YAML only because one format is always best
- **B.** Raw comma splitting, repr, and eval
- **C.** JSON, CSV, then arbitrary Python code
- **D.** CSV for rows, JSON for the snapshot, and safely loaded YAML for configuration

Write your choices before checking the answer key at the end of this book.

9. Key points for structured records

- CSV, JSON, and YAML solve overlapping but different interchange problems; choose from data shape, producers, consumers, and trust boundary.
- Use the CSV module for quoting and multiline records, open files with `newline=""`, validate headers, and convert field types explicitly.

- `json.loads`/`dumps` work with in-memory serialized values; `load`/`dump` work with file-like objects. JSON parsing is not domain validation or framing.
- Use `ensure_ascii=False` plus UTF-8 for readable multilingual JSON, and select formatting or key order when deterministic output matters.
- Use PyYAML's `safe_load` and `safe_dump`, but validate the resulting Python value just as carefully as data from another parser.
- Preserve source, logical record, raw values, and reason for anticipated rejected data. Let unexpected programming defects remain visible.

10. References and next step

- Python documentation: CSV file reading and writing
- Python documentation: JSON encoding and decoding
- PyYAML documentation: safe loading and dumping

Next, strengthen the validation stage. You will turn messy text into accepted records or explainable rejections by separating normalization, recognition, extraction, conversion, structural rules, and domain rules.

Turn Messy Text into Trustworthy Records

Online lesson and runnable notebook

Recognize and extract text patterns, validate structure and domain rules, and preserve accepted records, rejected evidence, and source lineage.

- **Level:** Python Foundations
- **Estimated time:** 4–5 hours
- **You will learn:** Prefer simple string operations when they fit, read a practical regular-expression subset, distinguish recognition from extraction, validate records in stages, choose fail-fast or error accumulation, and keep input lineage through a complete pipeline.
- **Practice in:** Google Colab, JupyterLab, or a local editor with messy beacon records

A radio observatory receives beacon lines written by several generations of equipment. A clean line looks like this:

```text
BCN-2042 | Europa Relay | temperature=18.5; battery=87
```

Other lines have surrounding spaces, an identifier with extra characters, a missing station name, a nonnumeric temperature, or a battery outside 0–100. A single regular expression can recognize punctuation, but it should not be asked to make every domain decision. This lesson builds a sequence whose stages can be explained and tested independently.

1. Start with the simplest string operations that fit

For exact prefixes, suffixes, delimiters, or whitespace policy, ordinary string methods are usually clearer than a regular expression:

```python
raw_line = "  BCN-2042 | Europa Relay | temperature=18.5; battery=87  "
line = raw_line.strip()

print(line.startswith("BCN-"))
print(line.count("|"))
print([part.strip() for part in line.split("|")])
```

This works if the contract is exactly three `|`-separated fields and the last field has a separately parsed structure. It becomes awkward if whitespace can vary within the punctuation or several identifier families are allowed.

Use a regular expression when you need a compact **pattern language**, not because it looks more advanced.

Need	Clear first choice
exact prefix or suffix	`startswith`, `endswith`

Need	Clear first choice
one literal delimiter	`split` or `partition`
remove known surrounding whitespace	`strip` under a stated policy
fixed character classes and repeated structure	regular expression
a full structured interchange format	its parser, not a regex

CSV quoting and JSON nesting are grammars with dedicated parsers. A regex should not replace them.

For a small, owned delimiter grammar, write the contract directly:

```python
def split_beacon_fields(raw_line):
    """Return three stripped beacon fields from an exact pipe-delimited line."""
    parts = [part.strip() for part in raw_line.strip().split("|")]
    if len(parts) != 3:
        raise ValueError(f"expected three pipe-delimited fields; got {len(parts)}")
    beacon_id, station, readings = parts
    if not all([beacon_id, station, readings]):
        raise ValueError("beacon fields must be non-empty")
    return beacon_id, station, readings


assert split_beacon_fields(
    "BCN-2042 | Europa Relay | temperature=18.5; battery=87"
) == ("BCN-2042", "Europa Relay", "temperature=18.5; battery=87")
```

This parser is easier to change if the only syntax is three pipe-separated fields. The later regular expression is justified by the more detailed numeric grammar. Comparing the two implementations is a design exercise, not a contest to use the most punctuation.

2. Read a useful regular-expression subset

Python's `re` module compiles a pattern and applies it to text. Raw string literals such as `r"\d"` keep backslashes from being interpreted first by the Python string-literal layer.

```python
import re

identifier_pattern = re.compile(r"BCN-\d{4}")

for candidate in ["BCN-2042", "BCN-42", "xxBCN-2042yy"]:
    print(candidate, bool(identifier_pattern.fullmatch(candidate)))
```

Read `BCN-\d{4}` from left to right:

- `BCN-` matches those literal characters;
- `\d` matches a Unicode decimal digit by default; and
- `{4}` requires exactly four repetitions of the preceding token.

If the external contract specifically means ASCII digits, express that as `[0-9]{4}` or use an ASCII flag. A pattern should encode the real contract, not a visually similar approximation.

Useful pieces for this unit:

Pattern piece	Meaning	Example
`.`	almost any single character	`A.B`
`[A-Z]`	one character in the range	uppercase ASCII letter
`[^|]`	one character except `|`	field content
`\s`	whitespace character	space, tab, or newline
`?`	zero or one repetition	optional sign
`*`	zero or more repetitions	possibly empty content
`+`	one or more repetitions	non-empty content
`{2,4}`	from two through four	bounded repetition
`(one|two)`	alternatives	either literal choice
`(...)`	capturing group	extract by position
`(?P<name>...)`	named group	extract by meaning
`^` and `$`	beginning and end anchors	whole-line intent

Raw strings do not make a pattern automatically correct. Compile small pieces, try accepted and rejected examples, and explain each token.

3. Choose `fullmatch`, `search`, or extraction deliberately

Three operations answer different questions:

```python
pattern = re.compile(r"BCN-[0-9]{4}")
text = "alert BCN-2042 received"

print("fullmatch:", pattern.fullmatch(text))
print("search:", pattern.search(text).group())
```

- `fullmatch` asks whether the *entire* string satisfies the pattern;
- `search` scans for a match anywhere; and
- `match` starts at the beginning but can leave an unmatched suffix; and
- `finditer` yields repeated matches with positions.

For a record identifier field, use `fullmatch`; accepting `"xxBCN-2042yy"` because it contains a valid-looking fragment violates the field contract. For scanning a log message that may mention an identifier, `search` is appropriate.

Named groups turn successful recognition into labeled extraction:

```python
BEACON_PATTERN = re.compile(
    r"(?P<beacon_id>BCN-[0-9]{4})\s*\|\s*"
    r"(?P<station>[^|]+)\s*\|\s*"
```

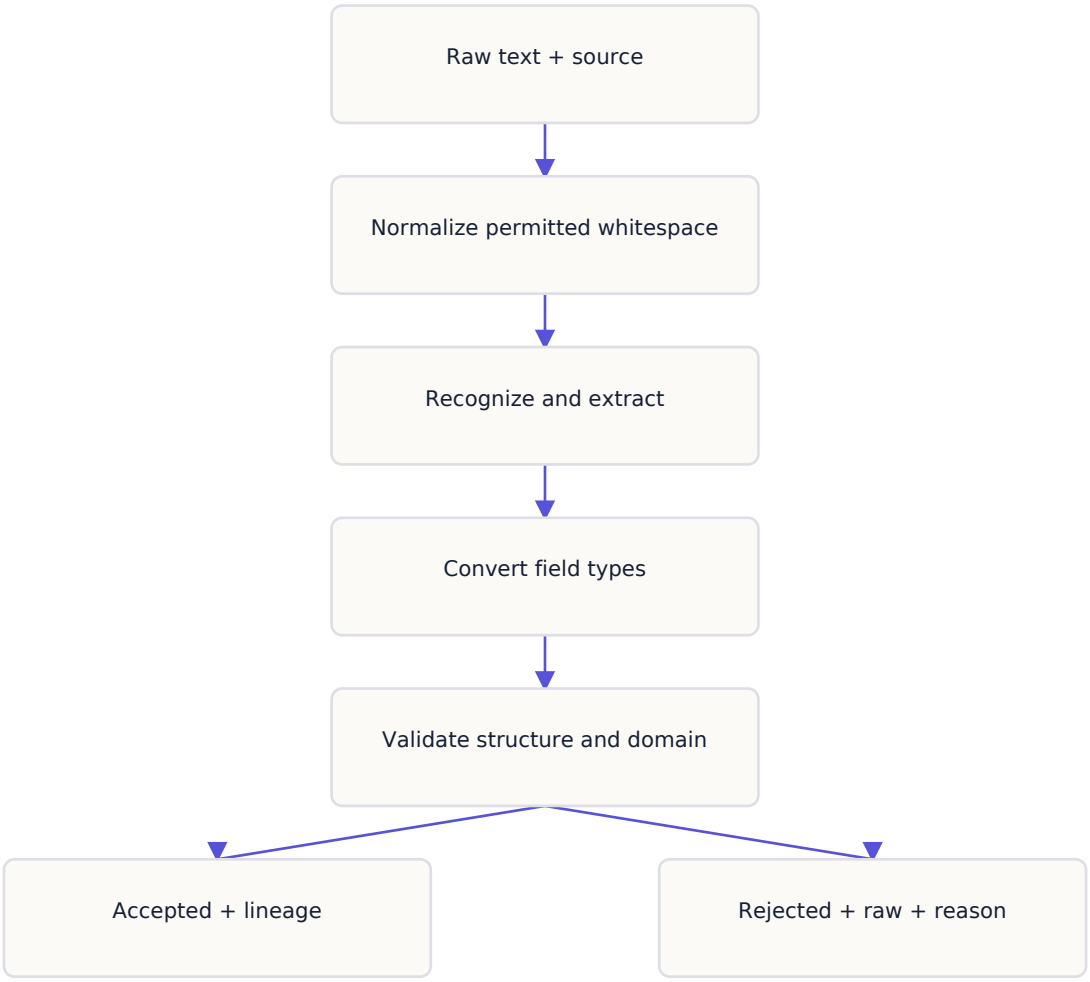
PYTHON (CONTINUED)

```
    r"temperature=(?P<temperature>[+-]?[0-9]+(?:\.[0-9]+)?);\s*"
    r"battery=(?P<battery>[0-9]+)"
)

candidate = "BCN-2042 | Europa Relay | temperature=18.5; battery=87"
match = BEACON_PATTERN.fullmatch(candidate)
assert match is not None
print(match.groupdict())
```

The pattern recognizes syntax and extracts **strings**. It does not prove that battery 187 is reasonable, temperature is within sensor limits, or the station is registered.

A trustworthy pipeline keeps boundary responsibilities separate and records where a candidate leaves the accepted path.



Validate structure and domain → Accepted + lineage: valid

Validate structure and domain → Rejected + raw + reason: anticipated failure

Unexpected programming exceptions leave the pipeline rather than being relabeled as bad input.

Checkpoint: use patterns for the right question

Match the operation to the text question

Choose the narrowest tool that expresses the stated contract.

1. A trusted line always contains exactly one literal `=` separating a key and value. What is a clear first tool?

- **A.** `partition("=")` plus explicit checks
- **B.** A CSV parser with an invented dialect
- **C.** `eval`
- **D.** A regex that accepts every character

2. Which operation should validate that an entire identifier field is exactly `BCN-` plus four digits?

- **A.** `search`, because extra prefixes should be ignored
- **B.** `splitlines`
- **C.** `fullmatch` with the identifier pattern
- **D.** `json.loads`

3. What has a successful named-group regex extraction of `battery=187` proven?

- **A.** That the battery is physically safe
- **B.** That the text matches the represented syntax; range validation is still needed
- **C.** That every required record field exists
- **D.** That no conversion can fail

Write your choices before checking the answer key at the end of this book.

4. Normalize only what the contract allows

Normalization makes equivalent permitted inputs share one representation. It should be **idempotent**: applying it twice yields the same result as once.

```python
def normalize_beacon_line(raw_line):
    """Remove surrounding whitespace while preserving internal field text."""
    return raw_line.strip()


sample = "  BCN-2042 | Europa Relay | temperature=18.5; battery=87  "
once = normalize_beacon_line(sample)
twice = normalize_beacon_line(once)
assert twice == once
```

Do not automatically collapse all internal whitespace, lowercase display names, or delete punctuation. Those transformations may change meaning or destroy forensic evidence. Preserve the original alongside any comparison form.

A record can carry lineage explicitly:

```python
candidate = {
    "source": "receiver-europa.log",
    "record_number": 17,
    "raw": sample,
    "normalized": normalize_beacon_line(sample),
}
```

Once data has passed through several functions, the source and record number are still available for explanations.

5. Convert values with chained context

After a successful match, convert fields under narrow exception boundaries:

```python
def convert_beacon_groups(groups, source, record_number):
    """Convert extracted beacon strings into typed values."""
    try:
        temperature = float(groups["temperature"])
    except ValueError as error:
        raise ValueError(
            f"{source}: record {record_number}: invalid temperature "
            f"{groups['temperature']!r}"
        ) from error

    try:
        battery = int(groups["battery"])
    except ValueError as error:
        raise ValueError(
            f"{source}: record {record_number}: invalid battery "
            f"{groups['battery']!r}"
        ) from error

    return {
        "beacon_id": groups["beacon_id"],
        "station": groups["station"].strip(),
        "temperature": temperature,
        "battery": battery,
    }
```

The regex already constrains these particular numeric strings, but keeping conversion checks close to the conversion makes the function robust if it is later reused with groups from another source. Exception chaining preserves the original failure.

6. Structural, field, and cross-field rules are distinct

A useful validator checks increasingly domain-specific claims:

```python
KNOWN_STATIONS = {"Europa Relay", "Luna South", "Mars Ridge"}


def validate_beacon(record):
    """Return record after checking field and cross-field rules."""
    required = {"beacon_id", "station", "temperature", "battery"}
    missing = required - record.keys()
    extra = record.keys() - required
    if missing:
        raise ValueError(f"missing fields: {sorted(missing)!r}")
    if extra:
        raise ValueError(f"unexpected fields: {sorted(extra)!r}")
```

PYTHON (CONTINUED)

```
    if not record["station"]:
        raise ValueError("station must be non-empty")
    if record["station"] not in KNOWN_STATIONS:
        raise ValueError(f"unknown station: {record['station']!r}")
    if not -120 <= record["temperature"] <= 80:
        raise ValueError("temperature outside sensor range -120..80")
    if not 0 <= record["battery"] <= 100:
        raise ValueError("battery must be between 0 and 100")
    if record["temperature"] < -80 and record["battery"] < 10:
        raise ValueError("extreme cold reports require battery of at least 10")

    return record
```

The sets of keys are structural rules. Station membership, ranges, and the relationship between cold temperature and battery are domain rules. A program can accumulate several messages for a form-like user experience, or fail at the first violation for a simple import API.

Some rules apply across records rather than inside one record. Duplicate IDs need batch state:

PYTHON

```
def require_unique_beacon_ids(records):
    """Return records after rejecting the first duplicated beacon identifier."""
    seen = set()
    for record in records:
        beacon_id = record["beacon_id"]
        if beacon_id in seen:
            raise ValueError(f"duplicate beacon_id: {beacon_id!r}")
        seen.add(beacon_id)
    return records


unique_preview = [
    {"beacon_id": "BCN-1001"},
    {"beacon_id": "BCN-1002"},
]
assert require_unique_beacon_ids(unique_preview) is unique_preview
```

Choose whether a duplicate rejects only the later record, both records, or the whole batch. That decision requires source order and ownership policy; a single-record regex cannot make it.

Choose and document one style:

PYTHON

```
def collect_battery_errors(value):
    """Return all independently checkable battery validation messages."""
    errors = []
    if not isinstance(value, int) or isinstance(value, bool):
        errors.append("battery must be an integer")
        return errors
    if value < 0:
        errors.append("battery cannot be negative")
    if value > 100:
        errors.append("battery cannot exceed 100")
    return errors
```

The early return avoids meaningless numeric comparisons after a type failure. Do not continue merely to produce more messages when later rules require a valid earlier stage.

Use a rule table before writing a large validator. It exposes dependencies and prevents an error message from claiming more than has been checked:

Order	Rule	Depends on	Failure category
1	value is a mapping with exact keys	parsed value	structure
2	identifier, station, and numeric fields have expected basic types	structure	field type
3	identifier follows the complete pattern	text identifier	recognition
4	station is known and ranges are satisfied	converted fields	domain
5	extreme cold has sufficient battery	valid temperature and battery	cross-field
6	identifier has not appeared earlier	accepted batch state	cross-record

Independent field rules can accumulate messages. Dependent rules should wait. For example, after `battery` fails the integer type check, “battery exceeds 100” is not a second useful diagnosis; comparison is not yet meaningful. That is why `collect_battery_errors` returns immediately after the type failure.

Error accumulation also changes the return contract. A function that raises one `ValueError` cannot simultaneously return a list of all field errors. Choose a consistent API such as `validate_or_raise(record)` for fail-fast code or `validation_messages(record)` returning a list for batch feedback, and name it so callers know which behavior to expect.

Checkpoint: keep validation stages explainable

Place each rule in the right stage

Choose the design that preserves meaning and evidence.

1. Why carry source and logical record number beside a candidate?

- **A.** To make strings encode themselves
- **B.** To avoid all validation
- **C.** To replace the raw input
- **D.** To connect an accepted or rejected result back to where it came from

2. What does idempotent normalization mean?

- **A.** Applying it twice produces the same result as applying it once
- **B.** It deletes every unusual character
- **C.** It catches every exception
- **D.** It writes two output files

3. Which is a cross-field rule?

- **A.** The record must be a dictionary

- **B.** Battery must parse as an integer
- **C.** When temperature is below -80, battery must be at least 10
- **D.** The input must have a source name

Write your choices before checking the answer key at the end of this book.

7. Compose the pipeline and quarantine expected rejections

One function can connect the stages without blending their responsibilities:

PYTHON
```
def parse_beacon_line(raw_line, source, record_number):
    """Return one validated beacon record or raise contextual ValueError."""
    normalized = normalize_beacon_line(raw_line)
    match = BEACON_PATTERN.fullmatch(normalized)
    if match is None:
        raise ValueError(
            f"{source}: record {record_number}: line does not match beacon syntax"
        )

    converted = convert_beacon_groups(match.groupdict(), source, record_number)
    return validate_beacon(converted)
```

A batch loader can quarantine anticipated `ValueError` instances:

PYTHON
```
def partition_beacon_lines(lines, source):
    """Return accepted records and detailed anticipated rejections."""
    accepted = []
    rejected = []

    for record_number, raw_line in enumerate(lines, start=1):
        try:
            record = parse_beacon_line(raw_line, source, record_number)
        except ValueError as error:
            rejected.append(
                {
                    "source": source,
                    "record": record_number,
                    "raw": raw_line,
                    "reason": str(error),
                }
            )
        else:
            accepted.append(
                {
                    **record,
                    "source": source,
                    "record": record_number,
                }
            )

    return accepted, rejected
```

Why not catch `Exception`? Examine a plausible defect:

PYTHON
```
def transform_beacon(record):
    """Return a display label for one parsed record."""
    return f"{record['beacon_id']} from {record['station']}"
```

PYTHON (CONTINUED)

```
try:
    transform_beacon(None)
except TypeError as error:
    print("programming failure remains visible:", error)
```

If a batch loop catches `Exception` and labels this as “invalid source data,” it hides a programming defect. Catch the exception type your parser deliberately uses for anticipated record rejection. Let `TypeError`, `KeyError` from an internal typo, and other unexpected defects stop the run with their traceback.

8. Decode the observatory beacon batch

Use these inputs:

PYTHON

```
beacon_lines = [
    "BCN-2042 | Europa Relay | temperature=18.5; battery=87",
    "xxBCN-2043yy | Luna South | temperature=20; battery=90",
    "BCN-2044 | Unknown Port | temperature=12; battery=70",
    "BCN-2045 | Mars Ridge | temperature=-95; battery=8",
    "BCN-2046 | Luna South | temperature=21; battery=101",
    "  BCN-2047 | Mars Ridge | temperature=-20; battery=65  ",
]
```

Write this prediction table before calling the pipeline:

Record	Expected stage	Expected result
1	all stages	accept `BCN-2042`
2	full-line recognition	reject wrapper text around the identifier
3	domain validation	reject the unknown station
4	cross-field validation	reject extreme cold with battery below 10
5	field range	reject battery 101
6	normalization then all stages	accept `BCN-2047`

For one accepted line, also run `split_beacon_fields`. Compare its three broad fields with the regex named groups. The split version is easier to read for the outer pipes; the regex earns its complexity by recognizing and extracting the numeric readings. A production parser could combine both rather than force one tool to express every layer.

Before running, predict which records are accepted and which stage rejects each other record. Then:

1. call `partition_beacon_lines` with source `receiver-night.log`;
2. assert accepted IDs are exactly `BCN-2042` and `BCN-2047`;
3. assert every rejection retains source, record, raw line, and non-empty reason;
4. assert `normalize_beacon_line` is idempotent for every input;

5 temporarily change `validate_beacon` to call a misspelled helper and confirm the resulting `NameError` is not quarantined as bad data;

6 restore the function and rerun from a clean state.

Check the expected partition and evidence after your clean run

```python
accepted, rejected = partition_beacon_lines(
    beacon_lines,
    "receiver-night.log",
)

assert [record["beacon_id"] for record in accepted] == ["BCN-2042", "BCN-2047"]
assert len(rejected) == 4
assert all(item["source"] == "receiver-night.log" for item in rejected)
assert all(isinstance(item["record"], int) for item in rejected)
assert all(item["raw"] in beacon_lines for item in rejected)
assert all(item["reason"] for item in rejected)
assert all(
    normalize_beacon_line(normalize_beacon_line(line))
    == normalize_beacon_line(line)
    for line in beacon_lines
)
```

Inspect the reasons instead of only counting them. One should describe syntax, one an unknown station, one the cross-field extreme-cold rule, and one battery range. That diversity proves several stages were exercised.

Checkpoint: preserve expected and unexpected failures

Keep a pipeline observable

Choose the exception and evidence policy that supports debugging.

1. Why does the batch loader catch only the `ValueError` used for anticipated record rejection?

- **A.** So unexpected programming failures keep their tracebacks instead of being mislabeled as data errors
- **B.** Because Python permits only one exception class
- **C.** So no record can ever be rejected
- **D.** Because `ValueError` closes every file

2. Which rejection record is most actionable?

- **A.** `{"bad": true}` only
- **B.** A count with no source
- **C.** Source, record number, raw value, and a specific reason
- **D.** The normalized value after deleting the original

3. Which order avoids domain checks on unconverted syntax?

- **A.** Serialize, delete, then recognize
- **B.** Validate every domain rule before parsing
- **C.** Catch Exception, then guess

- **D.** Normalize permitted differences, recognize/extract, convert, then validate

Write your choices before checking the answer key at the end of this book.

9. Key points for trustworthy validation

- Prefer clear string methods for simple delimiters and exact prefixes; use a regular expression when a pattern language genuinely clarifies the contract.
- `fullmatch` validates a whole field, while `search` finds a substring. Choose according to the question you are asking.
- Recognition, extraction, conversion, structural validation, field rules, and cross-field rules are separate responsibilities.
- Preserve original values and lineage. Make permitted normalization idempotent instead of gradually changing data on reruns.
- Choose fail-fast or accumulated errors based on how later rules depend on earlier success.
- Quarantine anticipated invalid records with useful evidence; do not convert unexpected programming failures into ordinary rejections.

10. References and next step

- Python documentation: regular-expression operations
- Python HOWTO: regular expressions
- Python tutorial: exception chaining and handling

Next, move below the text decoder. You will inspect raw byte values, file signatures, binary chunks, mutable byte buffers, and in-memory streams without pretending arbitrary binary data is encoded prose.

Cross the Text–Binary Boundary Safely

Online lesson and runnable notebook

Inspect byte sequences, use binary files and chunks, work with mutable buffers and in-memory streams, and decode only documented text payloads.

- **Level:** Python Foundations
- **Estimated time:** 4–5 hours
- **You will learn:** Read byte literals and hexadecimal evidence, distinguish byte indexing from slicing, use binary modes and signatures, process chunks correctly, mutate `bytearray`, control stream position, and decide when `memoryview` is worth using.
- **Practice in:** Google Colab, JupyterLab, or a local editor with synthetic telemetry packets

A probe sends a telemetry file containing a short ASCII signature, a version byte, a payload length, UTF-8 label bytes, and measurements. Only one portion is text. Opening the whole artifact as UTF-8 would confuse a binary container with a text document. This lesson keeps those roles visible.

Ask these questions whenever raw data appears:

- Does the format define this region as text, integers, compressed data, or opaque bytes?
- Which byte offsets and byte order does the specification promise?
- Is this operation using a text stream or a binary stream?
- Does a loop stop on `b""`, the binary end-of-stream marker?
- Do I need mutable bytes, an in-memory stream, or merely the original immutable value?
- Will avoiding one copy make a measurable difference here?

1. A `bytes` value is a sequence of small integers

Byte literals use a `b` prefix. ASCII-range printable values often appear as characters in their representation, while other values appear as hexadecimal escapes:

```python
packet = b"FC\x01\x00\xff"

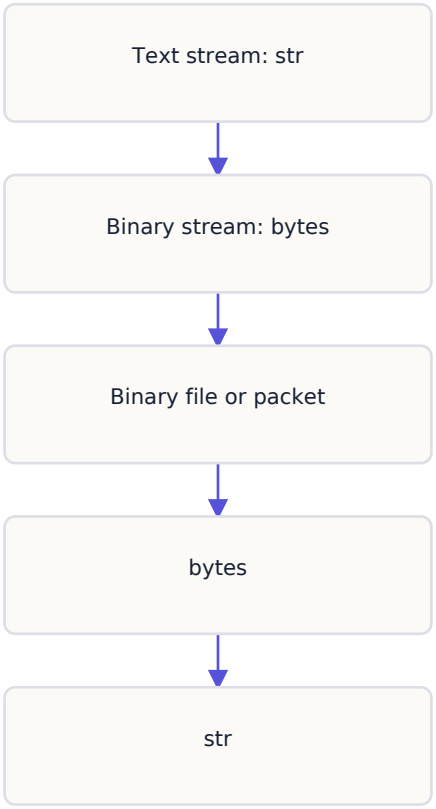
print(packet)
print("length:", len(packet))
print("first item:", packet[0])
print("first slice:", packet[:1])
print("hex:", packet.hex(" "))
```

`packet[0]` is the integer `70`, the byte value for ASCII `F`. `packet[:1]` is the one-byte value `b"F"`. Indexing and slicing intentionally return different types.

```python
assert packet[0] == 0x46
assert packet[:2] == b"FC"
assert bytes.fromhex("46 43 01 00 ff") == packet
```

Hexadecimal is a compact notation for byte values: `0xff` is decimal 255. It is not text that needs decoding.

Text and binary streams enforce different value types; encoding or decoding is allowed only where the format defines a text boundary.



Text stream: str → Binary stream: bytes: encode with declared codec

bytes → str: decode only documented text region

Arbitrary image, compressed, encrypted, or numeric bytes do not become meaningful by trying UTF-8.

A byte literal can directly contain only ASCII source characters. Encode Unicode text explicitly:

```python
label_text = "café-月"
label_bytes = label_text.encode("utf-8")
assert label_bytes.decode("utf-8") == label_text
print(label_bytes.hex(" "))
```

2. Text and binary streams reject the wrong value type

An in-memory demonstration makes stream type strictness easy to see:

```python
from io import BytesIO, StringIO

text_stream = StringIO()
binary_stream = BytesIO()

text_stream.write("signal")
```

PYTHON (CONTINUED)

```
binary_stream.write(b"signal")

try:
    text_stream.write(b"wrong type")
except TypeError as error:
    print("text stream:", error)

try:
    binary_stream.write("wrong type")
except TypeError as error:
    print("binary stream:", error)
```

Text streams accept and return `str`; binary streams accept and return bytes-like values. This early failure is useful because it forces the codec boundary to be explicit.

Checkpoint: identify byte values accurately

Know what a byte operation returns

Choose the exact type and boundary behavior.

1. What does `b"FC"[0]` return?

- **A.** The string `"F"`
- **B.** The integer byte value 70
- **C.** The one-byte value `b"F"`
- **D.** A file handle

2. When should part of a binary packet be decoded as UTF-8?

- **A.** Whenever any byte is above 127
- **B.** Only when the format contract defines that region as UTF-8 text
- **C.** After deleting every zero byte
- **D.** For every image and compressed file

3. What happens when code writes a `str` directly to `BytesIO`?

- **A.** It guesses an encoding
- **B.** It writes JSON
- **C.** It raises `TypeError`; the caller must encode deliberately
- **D.** It switches to text mode

Write your choices before checking the answer key at the end of this book.

3. Binary file modes preserve byte values

Use `rb`, `wb`, `ab`, or `xb` for binary files. `Path.read_bytes()` and `write_bytes()` are convenient whole-file operations:

PYTHON

```
from pathlib import Path
from tempfile import TemporaryDirectory
```

PYTHON (CONTINUED)

```
binary_temporary_directory = TemporaryDirectory()
workspace = Path(binary_temporary_directory.name)
packet_path = workspace / "probe.fct"
original = b"FCTM" + bytes([1, 3]) + "月".encode("utf-8") + b"\x00\xff"

written = packet_path.write_bytes(original)
restored = packet_path.read_bytes()

print("bytes written:", written)
print("signature:", restored[:4])
print("all bytes:", restored.hex(" "))
assert restored == original
```

No encoding or newline argument belongs to `read_bytes`; the operation does not interpret text.

Many binary formats begin with a **signature** or magic byte sequence. It is a useful early check, not complete validation:

PYTHON

```
EXPECTED_SIGNATURE = b"FCTM"


def require_telemetry_signature(data, source="<memory>"):
    """Raise when data does not start with the telemetry signature."""
    if not data.startswith(EXPECTED_SIGNATURE):
        found = data[:4].hex(" ")
        raise ValueError(
            f"{source}: expected signature {EXPECTED_SIGNATURE!r}; found {found}"
        )


require_telemetry_signature(restored, packet_path.name)
```

Two different formats can share a suffix, and a malicious or corrupted file can copy a signature. Continue validating version, lengths, checksums, and fields specified by the format.

4. Parse a small, documented packet layout

Use a deliberately simple teaching format:

Offset	Size	Meaning
0	4 bytes	signature `FCTM`
4	1 byte	format version
5	1 byte	UTF-8 label byte length
6	variable	label bytes
after label	remaining	opaque payload bytes

PYTHON

```
def parse_packet(data, source="<memory>"):
    """Return version, UTF-8 label, and opaque payload from an FCTM packet."""
    require_telemetry_signature(data, source)
    if len(data) < 6:
        raise ValueError(f"{source}: header is truncated")
```

PYTHON (CONTINUED)

```
    version = data[4]
    label_length = data[5]
    label_end = 6 + label_length
    if len(data) < label_end:
        raise ValueError(f"{source}: label is truncated")

    label_bytes = data[6:label_end]
    try:
        label = label_bytes.decode("utf-8")
    except UnicodeDecodeError as error:
        raise ValueError(f"{source}: label is not valid UTF-8") from error

    return {
        "version": version,
        "label": label,
        "payload": data[label_end:],
    }


parsed = parse_packet(restored, packet_path.name)
print(parsed)
assert parsed["version"] == 1
assert parsed["label"] == "月"
assert parsed["payload"] == b"\x00\xff"
```

Only the label slice is decoded. Version and length are already integers after indexing, while the payload remains opaque bytes.

5. Stream chunks until the binary end marker

Whole-file reads require memory proportional to file size. A chunk loop uses a bounded buffer:

PYTHON

```
def count_binary_bytes(path, chunk_size=4):
    """Return total byte count by reading positive-sized chunks."""
    if chunk_size <= 0:
        raise ValueError("chunk_size must be positive")

    total = 0
    with Path(path).open("rb") as handle:
        while True:
            chunk = handle.read(chunk_size)
            if chunk == b"":
                break
            total += len(chunk)
            print("chunk:", chunk.hex(" "))
    return total


assert count_binary_bytes(packet_path, chunk_size=3) == len(original)
```

At end of a binary stream, `read()` returns `b""`. Do not write `if chunk is None`; ordinary file reads do not use `None` for end-of-stream. Validate a positive chunk size so the loop cannot repeatedly request zero bytes and mistake that result for natural progress.

A chunk boundary is arbitrary. A multibyte UTF-8 character, numeric field, or compressed block can span chunks. Do not decode each arbitrary chunk independently. Use an incremental decoder or buffer complete format records when a streaming text protocol requires it.

Sometimes only a bounded header is needed. Read that contract instead of the entire attachment:

```python
def read_packet_header(path):
    """Return signature, version, and label length from a six-byte header."""
    with Path(path).open("rb") as handle:
        header = handle.read(6)
    if len(header) != 6:
        raise ValueError(f"{path}: expected a six-byte header; got {len(header)}")
    return header[:4], header[4], header[5]


signature, version, label_length = read_packet_header(packet_path)
assert signature == b"FCTM"
assert version == 1
assert label_length == 3
```

The label 月 is one character but three UTF-8 bytes, so the length field is 3. Header inspection can reject an unsupported version before allocating or reading a large body. If the format later uses multibyte integers, apply its documented byte order with tools such as `int.from_bytes` or `struct`, not decimal parsing.

Checkpoint: inspect and stream binary files

Respect binary layout and end-of-stream

Choose the behavior supported by the file-format contract.

1. What does a matching four-byte file signature prove?

- **A.** The entire file is valid and safe
- **B.** The suffix was renamed
- **C.** The initial bytes match one expected marker; remaining structure still needs validation
- **D.** The file is UTF-8 prose

2. What value marks end-of-stream for `handle.read(size)` on a normal binary file?

- **A.** `b""`
- **B.** `None`
- **C.** `-1`
- **D.** A newline string

3. Why can decoding every arbitrary byte chunk independently fail for valid UTF-8 text?

- **A.** UTF-8 has no bytes
- **B.** A multibyte character can be split across chunk boundaries
- **C.** Binary reads always add spaces
- **D.** Chunks automatically become JSON

Write your choices before checking the answer key at the end of this book.

6. Use `bytearray` when bytes must change in place

`bytes` is immutable. A `bytearray` is a mutable sequence of byte integers:

```python
corrupted = bytearray(b"XCTM\x01\x00")
print(corrupted)

corrupted[0] = ord("F")
corrupted.extend(b"OK")
repaired = bytes(corrupted)

print(repaired)
assert repaired == b"FCTM\x01\x00OK"
```

Mutation is useful for constructing packets, updating a known header field, or receiving data into a reusable buffer. It also creates aliasing concerns: two names can refer to the same mutable `bytearray`. Convert to `bytes` when you need an immutable snapshot.

```python
shared_buffer = bytearray(b"ABC")
alias = shared_buffer
snapshot = bytes(shared_buffer)

alias[0] = ord("Z")
assert shared_buffer == bytearray(b"ZBC")
assert snapshot == b"ABC"
```

The alias observes mutation because both names refer to one object. The bytes snapshot preserves the earlier state, reconnecting this binary example to Unit 6’s reference and copying rules.

Slice assignment can change length:

```python
buffer = bytearray(b"ABxxxxEF")
buffer[2:6] = b"CD"
assert buffer == bytearray(b"ABCDEF")
```

Validate offsets before mutation. One incorrect index can produce a syntactically valid but semantically corrupted artifact.

Every assigned integer must remain in the byte range:

```python
try:
    buffer[0] = 256
except ValueError as error:
    print(type(error).__name__, error)
```

Negative values and values above 255 are not byte values. Converting arbitrary integers with modulo arithmetic would hide an upstream range defect unless the binary format explicitly defines wrapping.

7. In-memory streams behave like files without filesystem I/O

`StringIO` holds text and `BytesIO` holds bytes. They are useful when an API expects a file-like object, when tests should not touch disk, or when a small artifact is assembled in memory.

```python
from io import BytesIO, StringIO

text_handle = StringIO()
text_handle.write("north\n")
text_handle.write("south\n")
```

PYTHON (CONTINUED)

```
print("cursor after writes:", text_handle.tell())

text_handle.seek(0)
print("first line:", repr(text_handle.readline()))
print("cursor after readline:", text_handle.tell())
print("complete value:", repr(text_handle.getvalue()))
text_handle.close()
print("closed?:", text_handle.closed)
```

`tell()` reports the current stream position. `seek(0)` moves back to the beginning. `getvalue()` returns the complete in-memory value regardless of the current cursor while the stream is open.

A closed stream enforces its resource boundary:

PYTHON

```
closed_stream = BytesIO(b"evidence")
closed_stream.close()

try:
    closed_stream.read()
except ValueError as error:
    print(type(error).__name__, error)
```

Retrieve needed values before closing, or use a `with BytesIO(...) as handle` block just as you would for a disk-backed handle.

Use `BytesIO` with APIs such as `json.dump` only after considering stream type: `json.dump` writes text and expects a text stream. To put JSON into a binary container, first create JSON text with `dumps`, then encode it under the container's documented codec.

The telemetry layout can also be assembled through a binary file-like API:

PYTHON

```
packet_stream = BytesIO()
packet_stream.write(b"FCTM")
packet_stream.write(bytes([1]))
stream_label = "Luna".encode("utf-8")
packet_stream.write(bytes([len(stream_label)]))
packet_stream.write(stream_label)
packet_stream.write(bytes.fromhex("00 ff"))

packet_from_stream = packet_stream.getvalue()
packet_stream.seek(0)
assert packet_stream.read(4) == b"FCTM"
assert parse_packet(packet_from_stream)["label"] == "Luna"
```

Each write advances the cursor, while `getvalue()` exposes the complete buffer. This is useful when a library accepts a binary handle even though the lesson or test should not create a real file.

8. `memoryview` can expose a buffer without copying its bytes

Slicing a `bytes` value creates another bytes object. `memoryview` provides a view over an object supporting the buffer protocol:

PYTHON

```
large_packet = b"HEAD" + bytes(range(64))
view = memoryview(large_packet)
payload_view = view[4:]
```

PYTHON (CONTINUED)

```
print(type(payload_view).__name__)
print(payload_view[:8].hex(" "))
print(payload_view.obj is large_packet)
```

The view refers to the original buffer. Convert with `bytes(payload_view)` only when an API needs an independent immutable bytes object.

Do not use `memoryview` merely because it sounds efficient. For small records, ordinary slicing is clearer and copying is negligible. Measure the actual workload before adding buffer-lifetime and mutability complexity. A view over a mutable buffer can observe later mutations, while some exporters prevent resizing until views are released.

9. Assemble and inspect a telemetry packet

Build a small packet without reading any external fixture:

PYTHON

```
def build_packet(label, payload, version=1):
    """Return an FCTM packet with one UTF-8 label and opaque payload."""
    if not 0 <= version <= 255:
        raise ValueError("version must fit in one byte")
    label_bytes = label.encode("utf-8")
    if len(label_bytes) > 255:
        raise ValueError("encoded label is too long")
    return b"FCTM" + bytes([version, len(label_bytes)]) + label_bytes + bytes(payload)
```

Then:

1. build a packet for label `"estação-月"` and payload `bytes.fromhex("00 10 ff 7f")`;
2. save it with `write_bytes` and verify the first four bytes in hexadecimal;
3. count it in chunks of 5 bytes;
4. parse it and compare the label, version, and exact opaque payload;
5. corrupt the signature and prove a contextual `ValueError` appears;
6. truncate the label and prove the length check detects it;
7. load the bytes into `BytesIO`, inspect `tell`, `seek`, and two reads;
8. create a `memoryview` of only the payload and explain whether avoiding a copy matters for this small packet.

Compare one complete packet round trip after your own clean run

PYTHON

```
from io import BytesIO
from pathlib import Path
from tempfile import TemporaryDirectory

payload = bytes.fromhex("00 10 ff 7f")
packet = build_packet("estação-月", payload)

with TemporaryDirectory() as temporary_name:
    path = Path(temporary_name) / "telemetry.fct"
    path.write_bytes(packet)

    assert path.read_bytes()[:4].hex(" ") == "46 43 54 4d"
    assert count_binary_bytes(path, chunk_size=5) == len(packet)
```

PYTHON (CONTINUED)

```
    restored = parse_packet(path.read_bytes(), path.name)
    assert restored == {"version": 1, "label": "estação-月", "payload": payload}

    try:
        parse_packet(b"FAIL" + packet[4:], "bad-signature.fct")
    except ValueError as error:
        assert "signature" in str(error)

    encoded_label_length = packet[5]
    truncated = packet[: 6 + encoded_label_length - 1]
    try:
        parse_packet(truncated, "truncated.fct")
    except ValueError as error:
        assert "truncated" in str(error)

    stream = BytesIO(packet)
    assert stream.tell() == 0
    assert stream.read(4) == b"FCTM"
    assert stream.tell() == 4
    stream.seek(0)
    assert stream.read() == packet

payload_view = memoryview(packet)[-len(payload):]
assert payload_view.tobytes() == payload
```

For four payload bytes, a normal slice is simpler. The view is an educational preview of an optimization to consider only after measurement.

Checkpoint: choose the appropriate buffer

Select a bytes tool for the job

Choose based on mutability, cursor behavior, and measured need.

1. When is `bytearray` more suitable than `bytes`?

- **A.** When a buffer must be changed in place under validated offsets
- **B.** When a string needs `.upper()`
- **C.** When JSON keys need sorting
- **D.** When every value must be hashable

2. What does `stream.seek(0)` normally do for a seekable `BytesIO` or `StringIO`?

- **A.** Closes the stream
- **B.** Moves its cursor to the beginning
- **C.** Changes bytes into text
- **D.** Deletes the buffer

3. When should a foundation-level program introduce `memoryview` into production code?

- **A.** For every two-byte value
- **B.** Whenever a variable needs a friendly name
- **C.** When measurement shows copying is meaningful and the buffer-lifetime complexity is understood

- **D.** To decode arbitrary images as UTF-8

Write your choices before checking the answer key at the end of this book.

10. Key points for binary boundaries

- `bytes` is an immutable sequence of integers from 0 through 255. Indexing returns an integer; slicing returns bytes.
- Text and binary streams enforce different value types. Encode and decode only where a format declares a text region and codec.
- Binary modes and `read_bytes`/`write_bytes` preserve exact byte values without encoding or newline translation.
- A signature, version, declared length, and payload each need their own checks; one matching marker is not complete validation.
- Chunk loops stop on `b""`, and arbitrary chunks do not necessarily align with text characters or format records.
- Use `bytearray` for deliberate in-place changes, `StringIO`/`BytesIO` for file-like in-memory APIs, and `memoryview` only for a measured copy cost.

11. References and next step

- Python documentation: binary sequence types
- Python documentation: `io` and in-memory streams
- Python documentation: `struct` for later binary layouts

You now have every boundary needed by the unit challenge: predictable discovery, UTF-8/BOM handling, CSV records, staged validation, narrow rejection handling, deterministic JSON, and verified replacement. Next, repair a starship archive whose code looks plausible but violates several of those contracts.

Unit Challenge: Restore the Starship Signal Archive

Online lesson and runnable notebook

Repair a multilingual multi-file import pipeline so it discovers signals predictably, rejects invalid records with evidence, and publishes verified JSON without hiding defects.

- **Level:** Python Foundations · Unit 9 challenge
- **Estimated time:** 60–120 minutes
- **Outcome assessed:** Discover and decode multiple files predictably, parse and validate CSV records, preserve rejection lineage, allow unexpected failures to propagate, and verify deterministic UTF-8 JSON before replacement.
- **Evidence:** Unchanged progressive assertions, four accepted multilingual signals, four detailed rejections, an unexpected-failure propagation check, a verified archive, a restored beacon phrase, and one debugging record

1. Reconnect a starship whose archive went silent

The research ship **Asteria** has emerged from a communications storm. Two signal files survived, but the archive builder now reverses their order, accepts an ID hidden inside extra characters, allows priority zero, calls programming defects “bad records,” and publishes JSON that violates the ship’s text contract.

The implementation is not blank. It looks reasonable and runs far enough to produce an archive—which is exactly why the defects are dangerous. Repair it one contract at a time until the accepted message fragments reveal:

```text
NOVA HELLO, LUNA SAFE HOME
```

You will work entirely inside a temporary starship workspace:

```text
starship-signals/
├── incoming/
│   ├── operator-notes.txt
│   ├── signal-alpha.csv
│   └── signal-beta.csv
└── restored/
    └── signal-archive.json   # created by your repaired pipeline
```

One CSV begins with a UTF-8 BOM. The records include accented and non-Latin names, a quoted multiline message, an invalid ID wrapper, an empty sender, a nonnumeric priority, and a priority outside the allowed range. The original inputs must remain unchanged.

Repair the archive, not the evidence

Keep the fixture and expected values unchanged. Run the nearest failing stage, write one hypothesis, make one focused change, and rerun all earlier stages. Hints are available, but first use the assertion message, exception type, serialized text, or path list already in front of you. :::

2. Read the restoration rules

File discovery

- Search only `incoming/signal-*.csv` beneath the supplied workspace root.
- Return `Path` objects in sorted, deterministic order.
- Do not include `operator-notes.txt`, a previous JSON output, or unrelated CSV files.
- Do not mutate or delete any input.

CSV and text boundary

- Open every signal file with `encoding="utf-8-sig"` and `newline=""`.
- Require headers in this exact order: `signal_id`, `sender`, `message`, `priority`.
- Enumerate logical data records from 1. A quoted newline remains part of one message and does not create an extra record.
- A header mismatch is a file-level `ValueError`; do not call it one rejected row.

Record validation

- `signal_id` must **fully match** `SIG-` followed by four ASCII digits.
- Strip surrounding whitespace from `signal_id`, `sender`, and `message`.
- `sender` and `message` must remain non-empty after stripping.
- Convert `priority` to an integer and require the inclusive range 1 through 5.
- When `int` raises `ValueError`, add source and record context with `raise ... from error`.
- Return a new normalized dictionary; do not modify the raw row retained as evidence.

Accepted and rejected records

- `load_signal_files` returns `(accepted, rejected)`.
- Catch only the contextual `ValueError` raised for an anticipated bad record.
- Each rejection stores `source`, `record`, `raw`, and `reason`.
- An unexpected `RuntimeError`, `TypeError`, or programming defect must stop the pipeline with its traceback.

Archive output

- Write one JSON document containing `accepted` and `rejected` lists.
- Use `ensure_ascii=False`, `indent=2`, and `sort_keys=True`.
- End the UTF-8 file with exactly one newline.
- Write a sibling `.tmp` artifact, parse it back, compare the restored value, and only then replace the destination.
- Remove a leftover temporary artifact if creation or verification fails.
- A second clean write with the same values must produce identical text.

3. Start from the contract

Run the fixture cell first. It creates a fresh workspace and all inputs, so the challenge does not depend on repository files or old notebook state.

PYTHON

```
import csv
import json
import re
from pathlib import Path
from tempfile import TemporaryDirectory


def create_starship_workspace(root):
    """Create and return a self-contained signal archive fixture beneath root."""
    root = Path(root) / "starship-signals"
    incoming = root / "incoming"
    restored = root / "restored"
    incoming.mkdir(parents=True)
    restored.mkdir()

    alpha_text = (
        "signal_id,sender,message,priority\n"
        "SIG-1001,Ana,NOVA,3\n"
        "xxSIG-1002yy,Luna,DRIFT,2\n"
        "SIG-1003,,QUIET,1\n"
        'SIG-1004,Renée,"HELLO,\nLUNA",5\n'
    )
    beta_text = (
        "signal_id,sender,message,priority\n"
        "SIG-2001,Noor,SAFE,2\n"
        "SIG-2002,Íris,DELAY,urgent\n"
        "SIG-2003,Kai,AGAIN,0\n"
        "SIG-2004,Zoë,HOME,4\n"
    )

    (incoming / "signal-alpha.csv").write_text(
        alpha_text,
        encoding="utf-8-sig",
        newline="",
    )
    (incoming / "signal-beta.csv").write_text(
        beta_text,
        encoding="utf-8",
        newline="",
    )
    (incoming / "operator-notes.txt").write_text(
        "Não arquivar automaticamente. 月 channel remains open.\n",
        encoding="utf-8",
    )
    (incoming / "unrelated.csv").write_text(
        "this,is,not,a,signal\n",
        encoding="utf-8",
    )
    return root


challenge_temporary_directory = TemporaryDirectory()
challenge_root = create_starship_workspace(challenge_temporary_directory.name)
```

Now run the starter unchanged. Preserve these five public names, signatures, and docstrings. It contains a small set of seeded defects; do not rewrite the whole pipeline before the assertions tell you where each promise breaks.

PYTHON

```
EXPECTED_SIGNAL_HEADERS = ["signal_id", "sender", "message", "priority"]
SIGNAL_ID_PATTERN = re.compile(r"SIG-[0-9]{4}")


def discover_signal_files(root):
    """Return intended signal CSV paths in deterministic order."""
    paths = list((Path(root) / "incoming").glob("signal-*.csv"))
    paths.reverse()
    return paths


def parse_signal(row, source, record_number):
    """Return one normalized signal or raise contextual ValueError."""
    signal_id = row["signal_id"].strip()
    sender = row["sender"].strip()
    message = row["message"].strip()

    if SIGNAL_ID_PATTERN.search(signal_id) is None:
        raise ValueError(
            f"{source}: record {record_number}: invalid signal_id {signal_id!r}"
        )
    if not sender:
        raise ValueError(f"{source}: record {record_number}: sender is empty")
    if not message:
        raise ValueError(f"{source}: record {record_number}: message is empty")

    try:
        priority = int(row["priority"])
    except ValueError as error:
        raise ValueError(
            f"{source}: record {record_number}: invalid priority {row['priority']!r}"
        ) from error

    if priority < 0 or priority > 5:
        raise ValueError(
            f"{source}: record {record_number}: priority must be 1 through 5"
        )

    return {
        "signal_id": signal_id,
        "sender": sender,
        "message": message,
        "priority": priority,
    }


def load_signal_files(paths):
    """Return accepted signals and detailed rejected-record evidence."""
    accepted = []
    rejected = []

    for path in paths:
        with Path(path).open(encoding="utf-8-sig", newline="") as handle:
            reader = csv.DictReader(handle)
            if reader.fieldnames != EXPECTED_SIGNAL_HEADERS:
                raise ValueError(
                    f"{path.name}: expected headers {EXPECTED_SIGNAL_HEADERS!r}; "
                    f"got {reader.fieldnames!r}"
                )

            for record_number, row in enumerate(reader, start=1):
                try:
                    signal = parse_signal(row, path.name, record_number)
                except Exception as error:
                    rejected.append(
                        {
                            "source": path.name,
                            "record": record_number,
```

PYTHON (CONTINUED)

```
                            "raw": dict(row),
                            "reason": str(error),
                        }
                    )
                else:
                    accepted.append(signal)

    return accepted, rejected


def write_signal_archive(path, accepted, rejected):
    """Safely write deterministic UTF-8 JSON and return destination path."""
    path = Path(path)
    path.parent.mkdir(parents=True, exist_ok=True)
    archive = {"accepted": accepted, "rejected": rejected}
    text = json.dumps(archive, indent=2, sort_keys=True)
    path.write_text(text, encoding="utf-8")
    return path
```

4. Repair one boundary at a time

Treat the archive as four cooperating subsystems:

Subsystem	Input	Success	Anticipated failure	Evidence
discovery	workspace root	sorted intended paths	none in supplied fixture	relative path list
record parser	raw row + lineage	normalized dictionary	contextual `ValueError`	value, message, `__cause__`
batch loader	discovered paths	accepted and rejected lists	record `ValueError` only	counts, raw rows, reasons
writer	lists + destination	verified deterministic JSON	serialization or filesystem failure	parsed output, exact text, final/tmp paths

Use this order:

1. make discovery select and sort the intended paths;
2. make identifier recognition consume the whole normalized field;
3. repair the lower priority boundary while preserving conversion context;
4. narrow the batch exception handler without losing rejections;
5. serialize readable Unicode and a final newline;
6. add temporary creation, parse-back verification, cleanup, and replacement;
7. run the complete archive twice from a clean state.

The same assertion may reveal a later defect after you fix an earlier one. That is progress, not regression.

Do not make the tests pass by weakening the mission

Do not edit expected filenames, accepted IDs, rejection counts, or the beacon phrase. If a check seems wrong, compare it with the written rule and fixture, then record the contradiction before changing an oracle. :::

5. Run progressive assertions

The helper below captures an anticipated validation failure and clearly fails if the action incorrectly succeeds:

```python
def captured_value_error(action):
    """Run action and return its ValueError, or fail if none is raised."""
    try:
        action()
    except ValueError as error:
        return error
    raise AssertionError("expected ValueError")
```

Stage A: discover only intended files in stable order

```python
paths = discover_signal_files(challenge_root)
relative_paths = [path.relative_to(challenge_root).as_posix() for path in paths]

assert relative_paths == [
    "incoming/signal-alpha.csv",
    "incoming/signal-beta.csv",
]
```

Run discovery again after creating an old output. Its result must not change:

```python
old_output = challenge_root / "restored" / "signal-archive.json"
old_output.write_text("previous good archive\n", encoding="utf-8")
assert discover_signal_files(challenge_root) == paths
```

Stage B: validate one record independently

```python
valid_row = {
    "signal_id": "  SIG-4242  ",
    "sender": "  航海士 月  ",
    "message": "  HOME  ",
    "priority": "5",
}
valid_snapshot = valid_row.copy()

assert parse_signal(valid_row, "focus.csv", 1) == {
    "signal_id": "SIG-4242",
    "sender": "航海士 月",
    "message": "HOME",
    "priority": 5,
}
assert valid_row == valid_snapshot
```

The whole identifier field must match:

```python
identifier_error = captured_value_error(
    lambda: parse_signal(
        {
            "signal_id": "xxSIG-4242yy",
            "sender": "Luna",
            "message": "DRIFT",
            "priority": "2",
```

PYTHON (CONTINUED)

```
        },
        "focus.csv",
        2,
    )
)
assert all(
    fragment in str(identifier_error)
    for fragment in ["focus.csv", "record 2", "signal_id"]
)
```

Check empty text and both priority failure families:

PYTHON

```
empty_sender_error = captured_value_error(
    lambda: parse_signal(
        {
            "signal_id": "SIG-4243",
            "sender": "   ",
            "message": "QUIET",
            "priority": "1",
        },
        "focus.csv",
        3,
    )
)
assert "sender" in str(empty_sender_error)

conversion_error = captured_value_error(
    lambda: parse_signal(
        {
            "signal_id": "SIG-4244",
            "sender": "Íris",
            "message": "DELAY",
            "priority": "urgent",
        },
        "focus.csv",
        4,
    )
)
assert "focus.csv" in str(conversion_error) and "record 4" in str(conversion_error)
assert isinstance(conversion_error.__cause__, ValueError)

zero_error = captured_value_error(
    lambda: parse_signal(
        {
            "signal_id": "SIG-4245",
            "sender": "Kai",
            "message": "AGAIN",
            "priority": "0",
        },
        "focus.csv",
        5,
    )
)
assert "1 through 5" in str(zero_error)
assert zero_error.__cause__ is None
```

Stage C: preserve CSV records and rejection lineage

```python
accepted, rejected = load_signal_files(paths)

assert [signal["signal_id"] for signal in accepted] == [
    "SIG-1001",
    "SIG-1004",
    "SIG-2001",
    "SIG-2004",
]
assert [signal["sender"] for signal in accepted] == [
    "Ana",
    "Renée",
    "Noor",
    "Zoë",
]
assert accepted[1]["message"] == "HELLO,\nLUNA"
assert [signal["priority"] for signal in accepted] == [3, 5, 2, 4]
assert len(rejected) == 4
assert [item["source"] for item in rejected] == [
    "signal-alpha.csv",
    "signal-alpha.csv",
    "signal-beta.csv",
    "signal-beta.csv",
]
assert [item["record"] for item in rejected] == [2, 3, 2, 3]
assert all(
    set(item) == {"source", "record", "raw", "reason"} and item["reason"]
    for item in rejected
)
assert (
    rejected[0]["raw"]["signal_id"],
    rejected[2]["raw"]["priority"],
) == ("xxSIG-1002yy", "urgent")
```

The multiline signal is one logical record. Its presence as accepted record 4 of the alpha file proves physical line breaks were not treated as separate CSV records.

Stage D: let an unexpected parser defect propagate

Temporarily replace the global parser, then restore it even if the experiment fails. The batch loader must not quarantine this `RuntimeError`.

```python
original_parse_signal = parse_signal


def crashing_parse_signal(row, source, record_number):
    raise RuntimeError("simulated navigation-computer defect")


parse_signal = crashing_parse_signal
try:
    try:
        load_signal_files(paths)
    except RuntimeError as error:
        assert "navigation-computer" in str(error)
    else:
        raise AssertionError("unexpected RuntimeError was hidden")
finally:
    parse_signal = original_parse_signal
```

Rerun Stage C after restoring the real parser.

Stage E: write, parse, and compare the archive

```python
archive_path = challenge_root / "restored" / "signal-archive.json"
written_path = write_signal_archive(archive_path, accepted, rejected)
archive_text = written_path.read_text(encoding="utf-8")
restored_archive = json.loads(archive_text)

assert written_path == archive_path
assert restored_archive == {"accepted": accepted, "rejected": rejected}
assert "Renée" in archive_text
assert "Zoë" in archive_text
assert "\\u00e9" not in archive_text.lower()
assert archive_text.endswith("\n")
assert not archive_text.endswith("\n\n")
assert not (archive_path.parent / "signal-archive.json.tmp").exists()
```

Prove the text is deterministic and a serialization failure preserves the previous good artifact:

```python
first_archive_text = archive_text
write_signal_archive(archive_path, accepted, rejected)
assert archive_path.read_text(encoding="utf-8") == first_archive_text

try:
    write_signal_archive(
        archive_path,
        accepted,
        [{"source": "broken", "raw": {"not-json-compatible"}}],
    )
except TypeError:
    pass
else:
    raise AssertionError("expected JSON serialization failure")

assert archive_path.read_text(encoding="utf-8") == first_archive_text
assert not (archive_path.parent / "signal-archive.json.tmp").exists()
```

Stage F: reveal the restored beacon

```python
beacon_phrase = " ".join(
    signal["message"].replace("\n", " ") for signal in accepted
)

assert beacon_phrase == "NOVA HELLO, LUNA SAFE HOME"
print(beacon_phrase)
```

Finally, restart the runtime, recreate the workspace, load your repaired five functions, and run every stage top to bottom. Confirm the fixture bytes and relative input names are unchanged.

6. Use the hint ladder only when needed

Hint 1

Map the first failed assertion to one boundary. A reversed relative-path list is only discovery. An accepted wrapper ID is only recognition. Priority zero is a domain boundary. A swallowed simulated `RuntimeError` points to the loader’s `except` clause. Escaped names or a missing final newline point to serialization.

Hint 2

The relevant operations already appeared in the lessons: `sorted`, regex `fullmatch`, an inclusive comparison, `except ValueError`, `json.dumps(..., ensure_ascii=False, indent=2, sort_keys=True) + "\n"`, and a sibling path whose name ends in `.tmp`. Apply one only where its contract belongs.

Hint 3

The core repair shapes are:

```python
paths = sorted((Path(root) / "incoming").glob("signal-*.csv"))

if SIGNAL_ID_PATTERN.fullmatch(signal_id) is None:
    raise ValueError(...)

if not 1 <= priority <= 5:
    raise ValueError(...)

try:
    signal = parse_signal(...)
except ValueError as error:
    rejected.append(...)

text = json.dumps(
    archive,
    ensure_ascii=False,
    indent=2,
    sort_keys=True,
) + "\n"
```

Write `text` to the sibling temporary path, load it with `json.load`, compare it to `archive`, then call `temporary_path.replace(path)`. Cleanup belongs in an exception path that immediately re-raises.

7. Keep debugging evidence

Preserve one failed stage that changed your understanding. The wrapper-ID check or simulated-runtime check makes a useful focused investigation.

Reproduction	Actual evidence	Hypothesis	One change	Focused rerun	Earlier stages	Clean rerun
exact call or assertion	path list, value, exception, or text	one mechanism that could be false	one expression or boundary	nearest stage result	pass/fail record	top-to-bottom result

A strong record is specific:

```text
Reproduction: parse_signal with signal_id "xxSIG-4242yy"
Expected: contextual ValueError
Actual: returned a normalized record
Hypothesis: search accepts a valid substring without checking the complete field.
Prediction: replacing search with fullmatch rejects the wrapper while SIG-4242 still passes.
```

Do not write “fixed regex.” Record the input, observed result, contract, hypothesis, and regression evidence. Preserve that record beside the final clean run.

8. Compare with a complete solution

Show the restored archive implementation after attempting every stage

PYTHON

```
EXPECTED_SIGNAL_HEADERS = ["signal_id", "sender", "message", "priority"]
SIGNAL_ID_PATTERN = re.compile(r"SIG-[0-9]{4}")


def discover_signal_files(root):
    """Return intended signal CSV paths in deterministic order."""
    return sorted((Path(root) / "incoming").glob("signal-*.csv"))


def parse_signal(row, source, record_number):
    """Return one normalized signal or raise contextual ValueError."""
    signal_id = row["signal_id"].strip()
    sender = row["sender"].strip()
    message = row["message"].strip()

    if SIGNAL_ID_PATTERN.fullmatch(signal_id) is None:
        raise ValueError(
            f"{source}: record {record_number}: invalid signal_id {signal_id!r}"
        )
    if not sender:
        raise ValueError(f"{source}: record {record_number}: sender is empty")
    if not message:
        raise ValueError(f"{source}: record {record_number}: message is empty")

    try:
        priority = int(row["priority"])
    except ValueError as error:
        raise ValueError(
            f"{source}: record {record_number}: invalid priority {row['priority']!r}"
        ) from error

    if not 1 <= priority <= 5:
        raise ValueError(
            f"{source}: record {record_number}: priority must be 1 through 5"
        )

    return {
        "signal_id": signal_id,
        "sender": sender,
        "message": message,
        "priority": priority,
    }


def load_signal_files(paths):
    """Return accepted signals and detailed rejected-record evidence."""
    accepted = []
    rejected = []

    for path in paths:
        with Path(path).open(encoding="utf-8-sig", newline="") as handle:
            reader = csv.DictReader(handle)
            if reader.fieldnames != EXPECTED_SIGNAL_HEADERS:
                raise ValueError(
                    f"{path.name}: expected headers {EXPECTED_SIGNAL_HEADERS!r}; "
                    f"got {reader.fieldnames!r}"
                )

            for record_number, row in enumerate(reader, start=1):
                try:
                    signal = parse_signal(row, path.name, record_number)
                except ValueError as error:
                    rejected.append(
                        {
                            "source": path.name,
                            "record": record_number,
                            "raw": dict(row),
                            "reason": str(error),
```

PYTHON (CONTINUED)

```
                        }
                    )
                else:
                    accepted.append(signal)

    return accepted, rejected


def write_signal_archive(path, accepted, rejected):
    """Safely write deterministic UTF-8 JSON and return destination path."""
    path = Path(path)
    path.parent.mkdir(parents=True, exist_ok=True)
    temporary_path = path.with_name(path.name + ".tmp")
    archive = {"accepted": accepted, "rejected": rejected}
    text = json.dumps(
        archive,
        ensure_ascii=False,
        indent=2,
        sort_keys=True,
    ) + "\n"

    try:
        temporary_path.write_text(text, encoding="utf-8", newline="\n")
        with temporary_path.open(encoding="utf-8") as handle:
            restored = json.load(handle)
        if restored != archive:
            raise ValueError("temporary archive did not round-trip")
        temporary_path.replace(path)
    except Exception:
        temporary_path.unlink(missing_ok=True)
        raise

    return path
```

Why these boundaries matter:

- sorted discovery makes file encounter order reproducible;
- `fullmatch` protects the complete identifier contract;
- the inclusive range rejects zero while conversion chaining retains the original `int` failure;
- the loader continues for anticipated invalid records but cannot hide a programming defect;
- the writer keeps multilingual names readable, creates exactly one document, verifies parsed values, and replaces only after success; and
- cleanup catches broadly only to remove its own temporary resource, then immediately re-raises the original failure.

Run every assertion in Section 5 unchanged after loading the solution.

9. Predict a changed mission rule

Asteria’s engineers propose archive contract version 3:

1. a third file, `signal-gamma.csv`, may arrive in UTF-8 **without** a BOM;
2. priority 5 signals must include a non-empty `acknowledged_by` column; and
3. the final archive must contain a `sources` list and a count by priority.

Before editing code, answer:

- Does `utf-8-sig` already handle a UTF-8 file without a BOM?
- Which exact header rule changes, and how will older files be versioned?
- Is `acknowledged_by` a field rule or a cross-field rule?
- Should counts be derived from accepted records or trusted from input?
- Which source order is deterministic?
- Which current assertions remain valuable regression evidence?
- What malformed gamma record would exercise the new rule without confusing it with CSV syntax?

Write new examples and expected values before implementing. Do not weaken the existing ID, sender, message, or priority contracts while adding a field.

10. Check your understanding

Starship archive restoration check

Connect each answer to the fixture, contract, and progressive evidence.

1. Why must `discover_signal_files` sort the intended matches?

- **A.** Sorting converts CSV to JSON
- **B.** The filesystem always returns reverse order
- **C.** It removes the need for validation
- **D.** Sorted paths make encounter order and derived output reproducible

2. Why does the identifier validator use `fullmatch` rather than `search`?

- **A.** The entire normalized field must satisfy the ID rule, not merely contain a matching substring
- **B.** Search cannot find digits
- **C.** Fullmatch decodes the CSV file
- **D.** It automatically converts priority

3. What should happen when `parse_signal` unexpectedly raises `RuntimeError`?

- **A.** Store it as an ordinary rejected row
- **B.** Delete the source file
- **C.** The error should propagate with its traceback
- **D.** Write an empty archive

4. Which combination keeps names such as Renée readable and the archive deterministic?

- **A.** ASCII ignore and random key order
- **B.** `repr` followed by append mode
- **C.** Manual string concatenation
- **D.** `ensure_ascii=False`, deterministic formatting, UTF-8, and a final newline

5. When should the temporary JSON artifact replace the current archive?

- **A.** Immediately after choosing its filename
- **B.** After it has been fully written, parsed back, and compared to the intended value
- **C.** Before JSON serialization
- **D.** Only after deleting every input

Write your choices before checking the answer key at the end of this book.

11. Decide whether the challenge is complete

Evidence rubric

Evidence	Ready to record when
Discovery	The unchanged path assertion returns only alpha then beta, even after output exists.
Parsing	Whole-field ID, non-empty text, conversion cause, and priority range checks pass without mutating raw rows.
CSV	The quoted multiline message remains one record and UTF-8/BOM inputs preserve names.
Boundary	Four anticipated rows have source, logical record, raw fields, and reasons; simulated `RuntimeError` propagates.
Output	Parsed JSON equals intended values, Unicode is readable, text is deterministic, exactly one final newline exists, and no `.tmp` remains.
Payoff	Accepted fragments reveal `NOVA HELLO, LUNA SAFE HOME`.
Reasoning	One debugging record connects exact evidence, one hypothesis, one controlled change, and focused/full reruns.
Reproducibility	A clean temporary workspace passes all progressive stages top to bottom.

This button stores a self-reported marker only in this browser. It does not submit work, grade it, verify identity, or issue a certificate.

Key points

- Stable discovery, explicit encoding/newline behavior, CSV parsing, and domain validation are distinct contracts in one reproducible import pipeline.
- Full-field recognition and inclusive range checks prevent plausible-looking invalid values from entering accepted records.
- Narrow record handling preserves bad-input evidence without hiding developer failures.
- Deterministic Unicode JSON becomes official only after a temporary artifact parses back to the intended value.

- The strongest restoration evidence is an unchanged top-to-bottom clean run, the complete rejection trail, and a beacon phrase derived from accepted data.

Unit 10: Modules, Environments, and Python Projects

Online lesson and runnable notebook

Move from isolated scripts to a versioned, installable Python project that works from a clean environment outside its repository.

- **Level:** Python Foundations · Unit 10
- **Estimated time:** 28–36 hours including the challenge
- **Unit outcome:** Find and evaluate reusable Python tools; split behavior across modules and packages with intentional imports and entry points; arrange and version a `src`-layout project; create and diagnose isolated environments; declare metadata and dependencies in `pyproject.toml`; and build, inspect, and clean-install a wheel whose public API works outside the repository.
- **Practice in:** Generated Colab notebooks for exploration and a local terminal project for multi-file and packaging work

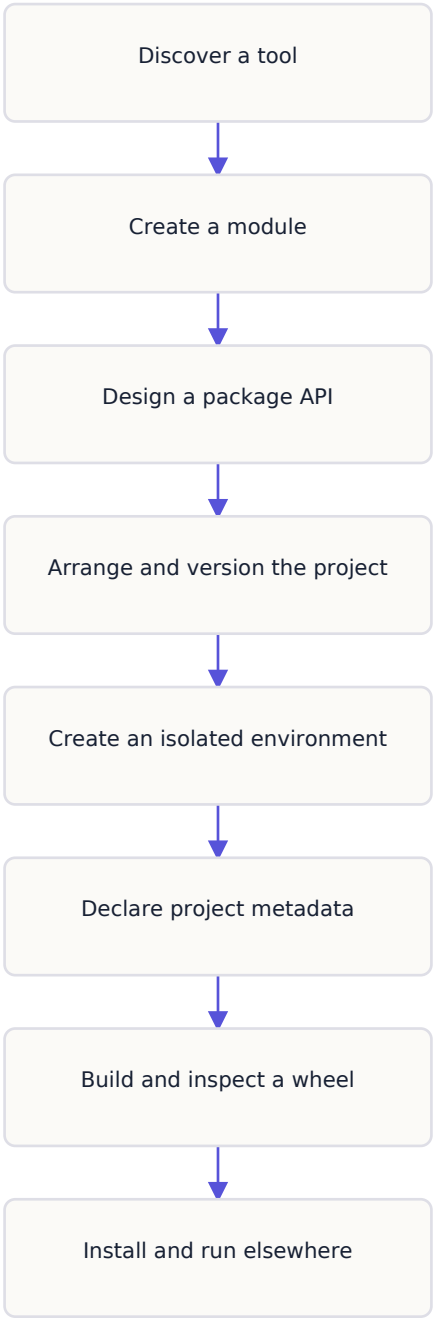
1. Grow a script into something another person can run

Until now, a notebook cell or one `.py` file has been enough for most tasks. A real project quickly develops more parts: reusable functions, a command that runs them, checks, documentation, configuration, dependencies, and version history. Merely putting those parts in folders does not make the result reliable. Python must be able to find the intended code, imports must avoid surprising work, collaborators need the same dependencies, and a built artifact must work away from the convenient repository directory.

Throughout this unit you will grow a small **constellation report** tool. It begins as a search for an existing standard-library function. It becomes a module, then a package with a deliberate public interface, then a versioned `src`-layout project. Finally, you will build a wheel and prove that a new environment can install and run it from another directory.

That last proof matters. “It worked in my project folder” may only mean that the current directory accidentally made source code importable. A clean installation asks a stronger question: **did the project declare, build, and ship everything its user actually needs?**

The unit follows one path from discovering a tool to verifying the artifact a different environment receives.



Each arrow adds a contract. A later step does not repair an unclear earlier one automatically.

2. Questions you will answer with evidence

Keep these questions nearby as you work:

- Is a capability built in, in the standard library, installed from a third-party distribution, or owned by this project?
- How can `help()`, signatures, examples, and primary documentation reveal a callable's contract?
- When does Python execute a module, where does it search, and why can an import succeed from one directory but fail from another?
- Which names should callers import, and which files are implementation details?

- What should run on import, with `python -m package`, or through an installed command?
- Which files belong in source, checks, documentation, configuration, generated output, and version history?
- Which exact interpreter will run a command, and which environment will receive an installation?
- Which dependencies are directly needed at runtime, needed only during development, or merely pulled in by another dependency?
- What do `[build-system]`, `[project]`, and `[project.scripts]` communicate?
- Can the built wheel import and run from a fresh environment outside the source tree?

3. Prepare the two practice spaces

Exploration that uses the standard library works in the generated Colab notebooks. Modules, Git, environments, and builds are clearer in a local folder and terminal because their behavior depends on real files and processes.

Before Lesson 2, confirm that these commands work locally:

```bash
python --version
python -m pip --version
git --version
```

On systems where the launcher is named `py`, use `py -m ...` consistently. On systems where Python 3 is named `python3`, use `python3 -m ...`. The important rule is not the spelling: use the **same intended interpreter** to create an environment and invoke its tools.

Create a disposable practice parent rather than experimenting inside an important repository:

```bash
mkdir unit10-practice
cd unit10-practice
```

Every lesson shows the expected working directory before shell commands. When a command changes files or installs software, read it before running and use the disposable workspace.

Colab is useful, but it is not the whole project lab

A hosted notebook can demonstrate imports and temporary files, but its runtime is replaced and its environment is managed for you. Complete the local project, Git, virtual-environment, build, and clean-install steps to observe the contracts this unit teaches. :::

4. Follow one connected sequence

Step	Lesson	What you will produce
1	Find the Right Tool Without Memorizing Everything	A repeatable discovery record that compares built-ins, standard-library tools, installed distributions, and project code.
2	Turn One Script into Reusable Modules	A multi-file constellation report with imports that behave differently from program entry points.
3	Design a Package People Can Rely On	A package facade whose public imports survive an internal file move.
4	Build a Project You Can Navigate and Share	A clear `src` project recorded in a small, inspectable Git history.
5	Give Each Project a Clean Python Environment	An isolated environment whose interpreter and installed distributions you can identify and recreate.
6	Build and Verify an Installable Python Project	Metadata, an editable installation, an sdist, a wheel, and a clean-install report.
7	Unit Challenge: Release the Moonlight Cipher Kit	A repaired puzzle package that reveals its phrase through both its public API and installed command.

The sequence is deliberate. Do not begin by copying a large `pyproject.toml` from an unfamiliar repository. First make module and package behavior explicit; then metadata can describe a project you understand.

5. Use four checkpoints instead of trusting the prompt

At each stage, record four kinds of evidence:

1. **Source evidence:** Which file owns the behavior, and which names form its public contract?
2. **Process evidence:** Which interpreter ran, from which working directory, with which arguments?
3. **Environment evidence:** Which environment and installed distribution versions were actually visible?
4. **Artifact evidence:** Which files entered the sdist or wheel, and did the installed artifact work away from the repository?

Prompt text such as `(.venv)` is a convenience for humans, not proof. An import that succeeds beside `src/` is not proof of installation. A successful build is not proof that the wheel contains the right files. Prefer commands that expose identity and behavior.

6. Suggested stopping points

This unit contains roughly 28–36 hours of active work. A useful pace is:

- **Session 1:** tool discovery and documentation experiments;
- **Sessions 2–3:** modules, import behavior, and execution boundaries;
- **Session 4:** package interfaces and import debugging;
- **Session 5:** project layout and a disposable Git history;
- **Session 6:** environment identity and dependency practice;
- **Sessions 7–8:** metadata, editable installation, building, inspection, and clean installation;
- **Session 9:** the Moonlight Cipher Kit challenge.

Stop after a lab passes from a fresh start, not merely when prose ends. Reopen a terminal in the project directory and repeat the proof command before moving on.

7. What success looks like

By the end, you should be able to receive a small script and turn it into a project without guessing:

- discover and justify reused functionality;
- split code without creating import-time surprises or cycles;
- state which package names callers may rely on;
- explain the repository tree to a collaborator;
- inspect what Git will record before committing;
- create a disposable environment and identify its interpreter;
- declare metadata and direct dependencies in one standards-based file;
- distinguish editable source from a built distribution;
- inspect wheel contents; and
- install the exact wheel into a new environment and run it outside the source checkout.

The later units will deepen classes, command-line interfaces, automated testing, quality tools, and documentation. Unit 10 supplies the project structure those tools will inhabit without trying to teach all of them early.

Continue to Find the Right Tool Without Memorizing Everything.

Find the Right Tool Without Memorizing Everything

Online lesson and runnable notebook

Discover, inspect, compare, and verify built-in, standard-library, installed, and project-owned Python tools without relying on memory or guesswork.

- **Level:** Python Foundations
- **Estimated time:** 3–4 hours
- **You will learn:** Search from the smallest dependency outward, inspect callable contracts, distinguish import packages from distributions, diagnose name shadowing, and justify whether to reuse or write a tool.
- **Practice in:** Google Colab, JupyterLab, or a local Python interpreter

A constellation report contains labels such as `" Vega-North "`. You need to remove surrounding spaces, normalize the case, count repeated labels, and find the most common one. You could write every operation yourself, but Python already supplies most of them. The professional skill is not memorizing a giant catalog. It is finding a plausible tool, reading its contract, and checking that the contract matches your data.

Answer these questions as you work:

- Where should a search begin before adding a dependency?
- What do a signature, docstring, example, and exception each reveal?
- How is `import yaml` related to a distribution named `PyYAML`?
- How can a local file silently replace the standard module you intended?
- When is a small project function a better choice than another dependency?

1. Search from the nearest tool outward

Use a **reuse ladder**. Stop at the first level that solves the complete problem clearly enough:

1. operations already available on the value, such as string methods;
2. built-in functions such as `len`, `sum`, `sorted`, and `enumerate`;
3. modules in Python’s standard library;
4. an already-declared project dependency;
5. a carefully evaluated new distribution; or
6. a small function owned and tested by the project.

The order reduces unnecessary code and dependencies, but it is not a command to force an awkward built-in solution. Readability and the full contract still matter.

```python
raw_labels = ["  Vega-North  ", "lyra-east", "VEGA-NORTH"]

normalized = [label.strip().casefold() for label in raw_labels]
print(normalized)
```

`strip` and `casefold` are methods already carried by each string. The list comprehension calls them in order for every label. `strip()` removes surrounding whitespace; `casefold()` performs an aggressive, Unicode-aware normalization for caseless matching. If the requirement were only display capitalization, `casefold()` would be the wrong contract even though the code runs.

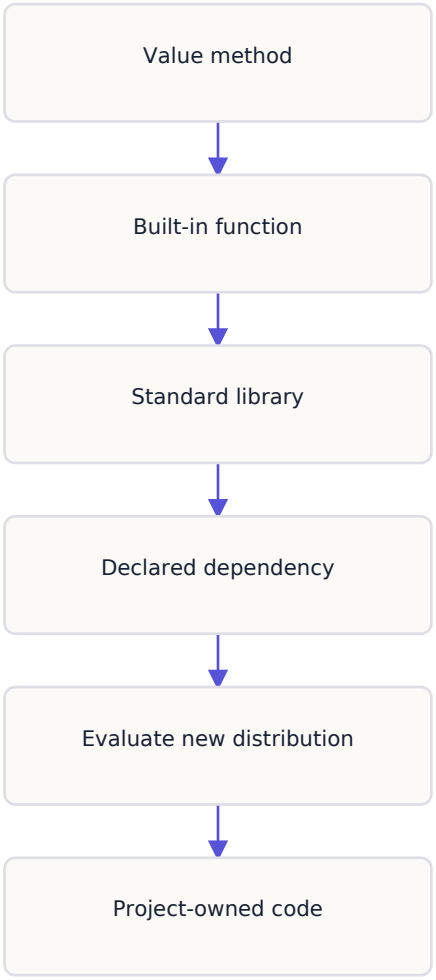
Now count without writing a loop that manages a dictionary manually:

```python
from collections import Counter

counts = Counter(normalized)
print(counts)
print(counts.most_common(1))
```

`collections` belongs to the standard library, and `Counter` is a dictionary subclass designed for counting hashable values. `most_common(1)` returns a list of up to one `(value, count)` pair—not the value by itself. That return shape is part of the contract.

Start close to the value and move outward only when the earlier level does not meet the requirement.



The final choice still needs a small verification example.

2. Inspect a callable before trusting its name

A useful name suggests intent, but it cannot state every accepted argument, return shape, or failure. Investigate in layers:

```python
import inspect

print(callable(str.removeprefix))
print(inspect.signature(str.removeprefix))
print(str.removeprefix.__doc__)
```

`callable(...)` answers whether Python can call the object. The signature shows that a string instance and a prefix are involved. The docstring gives a compact behavior description. In a notebook, `help(str.removeprefix)` presents related information in a pager or output cell:

```python
help(str.removeprefix)
```

Read parameter markers rather than skipping them. For example:

```python
import inspect

print(inspect.signature(sorted))
```

The signature is `sorted(iterable, /, *, key=None, reverse=False)`. The slash means arguments before it are positional-only. The star means `key` and `reverse` must be named. These calls therefore communicate different things:

```python
stars = ["Sirius", "sun", "Betelgeuse"]

print(sorted(stars))
print(sorted(stars, key=str.casefold))
print(sorted(stars, key=len, reverse=True))
```

The first uses default text ordering. The second compares caseless keys without changing the original strings. The third orders by length from greatest to least. `sorted` returns a new list; it does not rearrange `stars`.

Ask the tool one narrow question

Documentation reading becomes reliable when paired with a miniature experiment:

```python
samples = [
    ("north-star", "north-"),
    ("star-north", "north-"),
    ("north-north", "north-"),
]

for value, prefix in samples:
    print(value, "->", value.removeprefix(prefix))
```

This reveals three boundaries: only the beginning is considered, a missing prefix leaves the value unchanged, and exactly one matching prefix is removed. The experiment is not a substitute for documentation; it checks your reading against representative values.

Checkpoint: read a callable contract

Read signatures and return shapes

Choose the answer supported by the examples and contract.

1. What does `sorted(stars, key=len)` do to `stars`?

- **A.** It returns a new list and leaves `stars` unchanged.
- **B.** It permanently sorts `stars` in place.
- **C.** It converts every star to its length.
- **D.** It returns one longest string.

2. What does `*` before `key` in the signature of `sorted` tell you?

- **A.** The key is repeated many times.
- **B.** `key` must be supplied by name.

- **C.** The key must be a string.
- **D.** Everything after the star is optional positional input.

3. What is `"north-north".removeprefix("north-")`?

- **A.** `""`
- **B.** `"north-north"`
- **C.** `"north"`
- **D.** An exception

Write your choices before checking the answer key at the end of this book.

3. Follow documentation from overview to proof

Good documentation has layers. A module index helps you discover a candidate; an API reference defines parameters and return values; a tutorial or example shows composition; and a small local check confirms the part your project depends on.

Suppose the constellation report needs an average observation count:

```python
from statistics import mean

observations = [3, 4, 8, 5]
average = mean(observations)

print(average)
print(type(average).__name__)
```

Do not stop at "`mean` averages numbers." Ask about empty data and accepted numeric types. A controlled failure makes one boundary visible:

```python
from statistics import StatisticsError, mean

try:
    mean([])
except StatisticsError as error:
    print(type(error).__name__, error)
```

The caller must now decide whether an empty collection is invalid, means "no result," or should be handled earlier. The standard library cannot choose the application rule.

Record a dependency-sized note

Do not copy an entire reference page into project notes. Record the qualified name, the contract detail your code relies on, one representative example, one important boundary, and a link to the primary documentation. :::

Read exceptions as part of the API

Different parsing tools make different promises:

```python
from pathlib import Path
```

PYTHON (CONTINUED)

```
candidate = Path("reports") / "night.txt"
print(candidate.suffix)
print(candidate.with_suffix(".json"))
```

Try an invalid suffix separately:

PYTHON

```
from pathlib import Path

try:
    Path("night.txt").with_suffix("json")
except ValueError as error:
    print(type(error).__name__, error)
```

Knowing that the suffix must begin with a dot helps you validate or construct the value at the right boundary. Exceptions are not footnotes; they describe which inputs are outside the callable’s promise.

Compare neighboring tools with one discriminating example

Search results often offer several plausible names. Do not choose the shortest name or the first result. Find an input on which the contracts differ. For caseless matching, `lower()` and `casefold()` often agree on ASCII text:

PYTHON

```
for value in ["VEGA", "Lyra"]:
    print(value.lower(), value.casefold())
```

That example cannot distinguish them. A better probe contains text for which Unicode caseless matching matters:

PYTHON

```
value = "STRAßE"
print("lower:   ", value.lower())
print("casefold:", value.casefold())
```

The results are not interchangeable for every language. The documentation says `casefold` is intended to remove case distinctions and can be more aggressive than `lower`. That makes it a candidate for comparison keys, while `lower` may better preserve a display-oriented transformation. The project must still decide whether Unicode caseless matching is its real requirement.

Also compare mutation and return contracts, not only names:

PYTHON

```
labels = ["vega", "Orion", "lyra"]
new_order = sorted(labels, key=str.casefold)
returned = labels.sort(key=str.casefold)

print("new:", new_order)
print("original after sort:", labels)
print("sort returned:", returned)
```

`sorted` accepts any iterable and returns a new list. `list.sort` mutates one list and returns `None`. Both order these labels, but only one matches a contract that must preserve the caller’s original list. A discriminating example turns “these tools sound similar” into an observable design decision.

4. Separate import names from distribution names

Three categories are often confused:

- A **module** is one importable unit, often one `.py` file.
- An **import package** is an importable namespace that can contain modules and subpackages.
- A **distribution package** is a versioned artifact installed by a package installer. It can provide one or several import packages.

The spelling need not match. The distribution `PyYAML`, for example, commonly provides the import package `yaml`. Installation commands name distributions; `import` statements name modules or import packages.

Use `find_spec` to ask whether the import system can locate a name without executing that module:

```python
from importlib.util import find_spec

for import_name in ["json", "statistics", "not_a_real_star_module"]:
    specification = find_spec(import_name)
    print(import_name, "found" if specification else "missing")
```

A found specification says the import system found a candidate. It does not prove that the module is trustworthy, compatible with your use, or safe to import. To inspect installed distribution metadata, use `importlib.metadata`:

```python
from importlib.metadata import PackageNotFoundError, version

for distribution_name in ["pip", "definitely-not-a-distribution"]:
    try:
        print(distribution_name, version(distribution_name))
    except PackageNotFoundError:
        print(distribution_name, "not installed")
```

`version()` expects a **distribution** name. It does not accept every import name. When tooling has recorded the mapping, `packages_distributions()` can help connect the two:

```python
from importlib.metadata import packages_distributions

providers = packages_distributions()
print("pip is provided by:", providers.get("pip", []))
```

Some imports have no distribution mapping, especially project source that has not been installed. Treat an empty result as evidence to investigate, not as proof that the import is from the standard library.

Checkpoint: distinguish what Python imports from what pip installs

Keep module and distribution identities separate

Choose the most precise interpretation.

1. A project installs `PyYAML` and later runs `import yaml`. Which statement is correct?

- **A.** Both names must always match exactly.

- **B.** `yaml` is the version number.
- **C.** `PyYAML` is a built-in function.
- **D.** The distribution name and import-package name are different kinds of identity.

2. What does a non-`None` result from `find_spec("orbits")` establish?

- **A.** The import system found a candidate for that import name.
- **B.** The distribution is current and secure.
- **C.** The module has already executed successfully.
- **D.** The module belongs to the standard library.

3. Which identity does `importlib.metadata.version(...)` look up?

- **A.** A function parameter
- **B.** An installed distribution name
- **C.** A filename suffix
- **D.** A Git branch

Write your choices before checking the answer key at the end of this book.

5. Prove which file an import selected

Imagine a project file named `statistics.py` beside your program. `import statistics` may now load the project file instead of the standard-library module. This is **shadowing**: an earlier matching name hides the intended one.

The safest first question is “which file did Python load?”

```python
import statistics

print(statistics.__name__)
print(statistics.__file__)
```

Most source modules expose `__file__`. Some built-in or unusual module loaders do not, so diagnostic code can use `getattr(statistics, "__file__", None)`.

Create a disposable reproduction rather than adding a conflicting file beside your notebook:

```python
import subprocess
import sys
from pathlib import Path
from tempfile import TemporaryDirectory

with TemporaryDirectory() as temporary_name:
    workspace = Path(temporary_name)
    (workspace / "statistics.py").write_text(
        "origin = 'local shadow'\n",
        encoding="utf-8",
    )
    result = subprocess.run(
        [
            sys.executable,
            "-c",
```

PYTHON (CONTINUED)

```
            "import statistics; print(statistics.__file__); "
            "print(hasattr(statistics, 'mean'))",
        ],
        cwd=workspace,
        text=True,
        capture_output=True,
        check=True,
    )
    print(result.stdout)
```

The child process starts in the temporary workspace. Its first search location finds `statistics.py`; that file has no `mean`, so `hasattr` prints `False`. Renaming the local file to something project-specific, deleting its generated `__pycache__`, and restarting the process removes the ambiguity. Do not repair ordinary project imports by inserting guessed paths into `sys.path`; correct the name or installation contract.

Other useful origin checks are:

PYTHON

```
from importlib.util import find_spec

specification = find_spec("statistics")
print(specification.origin if specification else "not found")
```

If an import behaves differently in two terminals, record the current working directory, `sys.executable`, `sys.path`, the module's origin, and the exact command. "Python cannot find it" is a symptom; those values are evidence.

6. Evaluate a new dependency before adding it

A third-party distribution can save months of work, but every dependency adds an installation, compatibility, maintenance, and trust decision. Evaluate the specific project need:

Question	Evidence to collect
Does it solve the complete requirement?	API reference and a small representative experiment
Is its Python/platform support compatible?	Project metadata and supported-version policy
Is it maintained for the intended use?	Release history, issue activity, and maintainer guidance
Is the license acceptable for the project?	Declared license information and organizational policy
Is the dependency proportionate?	Compare its delivered capability with a clear project-owned function
Can the team replace or contain it?	Place it behind a small project interface when appropriate

Popularity alone is not a contract. Neither is the first search result. Prefer the project's primary documentation and package-index metadata, then verify the small slice of behavior your code needs.

For this unit's label normalization, string methods and `Counter` are enough. Adding a new distribution would increase setup work without improving the contract. A specialized astronomy coordinate conversion, however, may justify a mature domain library rather than a handwritten scientific implementation.

7. Build a tool-discovery dossier

Create a short record for this requirement:

Given labels containing surrounding spaces and mixed case, return each normalized label and the two most frequent labels. Equal counts should retain first-seen order.

Use these required names:

```python
from collections import Counter


def normalize_label(value):
    """Return a stripped, caseless label suitable for matching."""
    return value.strip().casefold()


def summarize_labels(values, limit=2):
    """Return normalized labels and their most common `(label, count)` pairs."""
    normalized = [normalize_label(value) for value in values]
    return normalized, Counter(normalized).most_common(limit)
```

Run ordinary and boundary checks:

```python
labels, leaders = summarize_labels(
    [" Vega ", "LYRA", "vega", "Orion", "lyra", "vega"]
)

assert labels == ["vega", "lyra", "vega", "orion", "lyra", "vega"]
assert leaders == [("vega", 3), ("lyra", 2)]
assert summarize_labels([], limit=2) == ([], [])
assert summarize_labels([" A ", "b"], limit=0) == (["a", "b"], [])
print(leaders)
```

Your dossier should contain:

1. the reuse-ladder level for `strip`, `casefold`, and `Counter`;
2. the signature or documented contract detail each use depends on;
3. one boundary learned from an experiment;
4. why no new distribution is justified; and
5. a primary-documentation link.

Changed requirement

Suppose display spelling must be preserved while matching remains caseless. Do not overwrite every label with its casefolded form. Change the representation so comparison keys and first-seen display labels remain available. Explain why the original return contract can no longer satisfy both needs. :::

Compare one possible display-preserving extension

```python
from collections import Counter


def most_common_display_labels(values, limit=2):
    """Return first-seen display labels paired with caseless counts."""
    display_by_key = {}
```

PYTHON (CONTINUED)

```
    keys = []
    for value in values:
        display = value.strip()
        key = display.casefold()
        keys.append(key)
        display_by_key.setdefault(key, display)

    counts = Counter(keys)
    return [
        (display_by_key[key], count)
        for key, count in counts.most_common(limit)
    ]


assert most_common_display_labels([" Vega ", "VEGA", "Lyra"]) == [
    ("Vega", 2),
    ("Lyra", 1),
]
```

The normalized key supports matching; `display_by_key` retains the first trimmed spelling. One value no longer has to serve two incompatible purposes.

Checkpoint: choose and verify a tool

Make a proportionate reuse decision

Choose the decision supported by the stated contract.

1. `statistics.mean` **is unexpectedly missing. What is the most useful first diagnostic?**

- **A.** Install several statistics distributions.
- **B.** Add the repository root to `sys.path`.
- **C.** Print `statistics.__file__` to identify what was imported.
- **D.** Rename every Python file.

2. String methods and `Counter` **completely satisfy a small label-counting contract. What is the best default?**

- **A.** Add the most popular text framework anyway.
- **B.** Rewrite `Counter` from scratch.
- **C.** Avoid standard-library imports.
- **D.** Use the existing tools and record the contract details relied on.

3. Why pair primary documentation with a miniature experiment?

- **A.** The documentation defines the contract, while the experiment checks your reading on relevant values.
- **B.** Experiments prove every possible behavior.
- **C.** Documentation is never current.
- **D.** The pair prevents all future dependency changes.

Write your choices before checking the answer key at the end of this book.

8. Key points for finding reusable tools

- Search value methods, built-ins, and the standard library before adding a new dependency, while still choosing for the complete requirement.
- Read signatures, parameter markers, docstrings, return shapes, and documented exceptions; verify the relevant behavior with a small example.
- A module or import package is what Python imports. A distribution is what an installer installs and versions; their names need not match.
- Use `find_spec`, `importlib.metadata`, `sys.executable`, and module origin as diagnostic evidence rather than guessing.
- A local file can shadow the intended module. Identify its origin before changing paths or installing packages.
- New dependencies deserve a proportional compatibility, maintenance, license, trust, and replacement decision.

References

- Python built-in functions and `help`
- Python standard library overview
- `inspect.signature` and callable introspection
- `importlib.util.find_spec`
- Accessing installed distribution metadata
- PyPA explanation of distribution packages and import packages

Turn One Script into Reusable Modules

Online lesson and runnable notebook

Split a working script into modules, trace import search and caching, diagnose import failures, and keep reusable behavior separate from program entry points.

- **Level:** Python Foundations
- **Estimated time:** 4–5 hours
- **You will learn:** Create and import real modules, trace namespaces and the import cache, inspect search locations and failures, and design a `main` boundary that supports both reuse and execution.
- **Practice in:** A local editor and terminal; temporary workspaces also run in Colab or JupyterLab

Your constellation report has grown into one long script. Coordinate conversion, label formatting, report assembly, demo data, and printing are mixed together. Changing one part means scrolling through all the others, and importing one useful function would run the whole report. This lesson separates those responsibilities without turning every function into its own microscopic file.

Keep five questions visible:

- What namespace does each import form create in the caller?
- Which statements run the first time a module is imported?
- Why does editing a module not automatically change an already imported object?
- Which search location supplied the module?
- How can one file offer reusable functions and still run as a program?

1. A module is an executed file with its own namespace

Create a folder named `constellation-report` containing this file:

PYTHON
```
# orbit_math.py
DEGREES_IN_CIRCLE = 360


def normalize_degrees(value):
    """Return an angle in the half-open range 0 <= result < 360."""
    return value % DEGREES_IN_CIRCLE
```

A `.py` file is a **module source file**. When Python imports it, Python creates a module object, gives that object a namespace, executes the file's top-level statements in that namespace, and caches the module object.

Add a second file beside it:

PYTHON
```
# report.py
import orbit_math
```

PYTHON (CONTINUED)

```
angles = [-15, 20, 390]
normalized = [orbit_math.normalize_degrees(angle) for angle in angles]
print(normalized)
```

From inside `constellation-report`, run:

BASH

```
python report.py
```

The expected output is:

TEXT

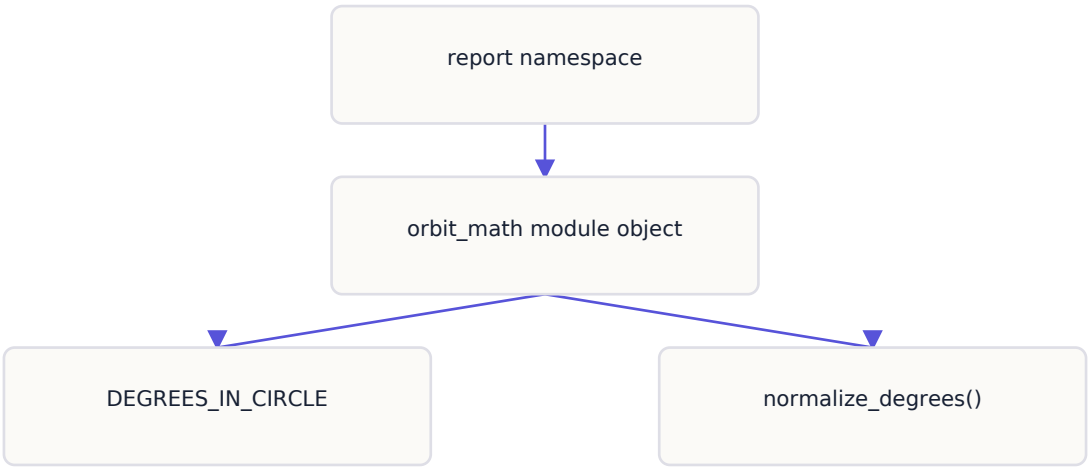
```
[345, 20, 30]
```

Trace the important names:

Namespace	Name	Value or role
`orbit_math` module	`DEGREES_IN_CIRCLE`	`360`
`orbit_math` module	`normalize_degrees`	function object
`report` module	`orbit_math`	module object
`report` module	`angles`	input list
`report` module	`normalized`	output list

`import orbit_math` does not copy every module name into `report`. It binds one name, `orbit_math`, and attribute access crosses the visible module boundary. That explicit qualifier helps a reader locate ownership.

An import binds a module object in the caller while the imported definitions remain in the module's own namespace.



report namespace → orbit_math module object: orbit_math

Top-level statements run during import

This version has an unwanted surprise:

```python
# noisy_orbits.py
print("Loading demo observations...")


def normalize_degrees(value):
    return value % 360
```

Any importer displays the message even if it only wants the function. Imports must execute definitions, assignments, and import statements needed to create the module. Avoid unrelated printing, input prompts, file writes, network calls, and expensive demo work at import time.

Checkpoint: trace module namespaces

Follow names across a module boundary

Choose what Python creates or executes.

1. After `import orbit_math`, which name is bound directly in the caller?

- **A.** `normalize_degrees`
- **B.** `orbit_math`
- **C.** `DEGREES_IN_CIRCLE`
- **D.** Every public name in the file

2. What happens to the top-level `print` in `noisy_orbits.py` on its first ordinary import?

- **A.** Python ignores every statement except `def`.
- **B.** It runs only when the function is called.
- **C.** It executes while Python initializes the module.
- **D.** It becomes the module docstring.

3. Why can `report.py` and `orbit_math.py` both use a name such as `value` safely?

- **A.** Names are always global across every file.
- **B.** Python renames every variable.
- **C.** Only uppercase names belong to modules.
- **D.** Each module has its own global namespace.

Write your choices before checking the answer key at the end of this book.

2. Choose an import form that keeps ownership clear

Python offers several import forms. They bind different names:

```python
import statistics
from pathlib import Path
from collections import Counter as Tally
```

PYTHON (CONTINUED)

```
print(statistics.mean([2, 4, 6]))
print(Path("reports") / "night.txt")
print(Tally("vega lyra vega".split()))
```

- `import statistics` binds the module name and keeps calls qualified.
- `from pathlib import Path` binds one selected attribute directly.
- `as Tally` binds an alias. Use aliases for established conventions or a real collision, not merely to invent shorter spellings.

Avoid star imports in application code:

PYTHON

```
# Avoid: the reader cannot see which names this statement introduces.
# from orbit_math import *
```

They obscure ownership, make collisions easy, and turn a dependency's added public name into a possible local behavior change.

Imports bind objects at a moment in time

Compare module-qualified access with a directly imported name:

PYTHON

```
import types

tools = types.SimpleNamespace(scale=2)
scale_copy = tools.scale
tools.scale = 3

print(tools.scale)
print(scale_copy)
```

The direct binding `scale_copy` still refers to the earlier integer. A `from module import name` statement similarly binds the object available at import time; it is not a permanently live link to later rebinding in the module.

For ordinary constants and functions this is usually fine. During debugging, however, it explains why changing or rebinding a module attribute does not update every name copied elsewhere.

3. The import cache prevents repeated initialization

Python records initialized modules in `sys.modules`. A later import of the same name normally returns that cached object rather than executing the source file again.

Use a child process and temporary file so the demonstration starts clean:

PYTHON

```
import subprocess
import sys
from pathlib import Path
from tempfile import TemporaryDirectory

with TemporaryDirectory() as temporary_name:
    workspace = Path(temporary_name)
    (workspace / "beacon.py").write_text(
```

PYTHON (CONTINUED)

```
        "print('initializing beacon')\nstatus = 'ready'\n",
        encoding="utf-8",
    )
    program = "import beacon; import beacon; print(beacon.status)"
    result = subprocess.run(
        [sys.executable, "-c", program],
        cwd=workspace,
        text=True,
        capture_output=True,
        check=True,
    )
    print(result.stdout)
```

Expected output:

TEXT

```
initializing beacon
ready
```

The statement appears once, not twice. The cache supports shared module state and avoids repeated setup. It also creates a notebook trap: after editing a source file, running `import beacon` again in the same kernel may reuse the old module. Restarting the kernel or process gives the cleanest beginner proof. `importlib.reload` exists, but existing `from ... import ...` bindings and created objects make reload behavior more subtle than “restart the file.”

Inspect the cache without mutating it:

PYTHON

```
import json
import sys

print("json" in sys.modules)
print(sys.modules["json"] is json)
```

Deleting arbitrary cache entries is not a normal repair strategy. Find the source of stale state and reproduce in a fresh process.

Checkpoint: reason about forms and caching

Choose imports and explain repeated behavior

Use namespace and cache behavior, not guesswork.

1. Which form most clearly shows that `mean` belongs to `statistics` at each call site?

- **A.** `import statistics` followed by `statistics.mean(...)`
- **B.** `from statistics import *`
- **C.** Copy the function into the caller
- **D.** Rename it to `m`

2. Why does a top-level message usually appear once after two imports of the same module in one process?

- **A.** `print` detects duplicates.
- **B.** The second import normally reuses the module in `sys.modules`.

- **C.** Python deletes the second import statement.
- **D.** Only constants can execute twice.

3. You edit an imported file, but the notebook still shows old behavior. What is the clearest beginner verification?

- **A.** Add several entries to `sys.path`.
- **B.** Use a star import.
- **C.** Restart the kernel and run from the top.
- **D.** Rename the function on every run.

Write your choices before checking the answer key at the end of this book.

4. Python searches specific locations in order

An import name is not a filesystem path. Python asks configured import finders to locate it. For ordinary source imports, `sys.path` displays the search path:

```python
import sys

print("interpreter:", sys.executable)
for position, location in enumerate(sys.path[:5]):
    print(position, repr(location))
```

The exact entries depend on how Python started, the environment, the platform, and configuration. The script directory or current working directory commonly appears near the beginning, followed by standard-library and environment locations. This is why project files can shadow installed or standard modules.

Use the import system to report a candidate origin:

```python
from importlib.util import find_spec

for name in ["pathlib", "orbit_math"]:
    specification = find_spec(name)
    print(name, specification.origin if specification else "not found")
```

In a notebook without `orbit_math.py` on its search path, the second result is `not found`. That is expected—not evidence that Python itself is broken.

Classify common import failures

These failures point to different places:

Symptom	Likely question
`ModuleNotFoundError: No module named 'orbits'`	Is the top-level name installed or on the search path for this interpreter?
`ImportError: cannot import name 'scale' from 'orbits'`	Was the module found but the selected attribute absent or not yet initialized?
Attribute missing from an unexpected local path	Is a file or package shadowing the intended module?

Symptom	Likely question
Import works only from the repository root	Is the code relying on the working directory instead of installation?
Import behaves differently after edits	Is an old module cached in this process?

Do not automatically append parent directories to `sys.path` inside project source. That hides a broken project or command contract and makes behavior depend on a guessed folder depth. Later in this unit, a `src` layout plus an editable installation will make the package available intentionally.

Separate discovery failure from module-execution failure

`ModuleNotFoundError` does not always mean the first import name was absent. Read the traceback and the exception's `name`:

```python
try:
    import definitely_missing_constellation
except ModuleNotFoundError as error:
    print("missing name:", error.name)
```

Now imagine `constellation` was found but its source contains `import missing_catalog`. The traceback begins while executing `constellation`, but the missing name is `missing_catalog`. Installing another copy of `constellation` does not address that dependency.

An import can also find a file and then fail during its top-level execution:

```python
# broken_catalog.py
DEFAULT_SCALE = 0


def adjusted(value):
    return value / DEFAULT_SCALE


# Avoid top-level demonstration work like this:
# preview = adjusted(10)
```

If the commented preview ran, discovery would have succeeded and initialization would have raised `ZeroDivisionError`. Preserve the full traceback: the exception category and failing source line distinguish "not found" from "found but broken while importing."

Bytecode caches are outputs, not source ownership

Python may create `__pycache__/` and `.pyc` files after imports. They store bytecode tagged for an interpreter implementation/version so later starts can avoid recompiling unchanged source. They do not replace the `.py` source as the project's maintained contract.

```python
import importlib.util

print(importlib.util.cache_from_source("orbit_math.py"))
```

The exact tag varies, which is one reason caches do not belong in Git. Python normally checks cache validity against source metadata. When a rename or unusual tool leaves the diagnosis confusing,

remove generated caches and reproduce in a fresh process—but do not treat routine cache deletion as the repair for an incorrect import graph.

Reproduce two search locations without changing the parent notebook

This lab proves that the same import spelling can select different files:

PYTHON
```
import subprocess
import sys
from pathlib import Path
from tempfile import TemporaryDirectory

with TemporaryDirectory() as temporary_name:
    root = Path(temporary_name)
    first = root / "first"
    second = root / "second"
    first.mkdir()
    second.mkdir()
    (first / "catalog.py").write_text("station = 'north'\n", encoding="utf-8")
    (second / "catalog.py").write_text("station = 'south'\n", encoding="utf-8")

    for working_directory in [first, second]:
        result = subprocess.run(
            [
                sys.executable,
                "-c",
                "import catalog; print(catalog.station); print(catalog.__file__)",
            ],
            cwd=working_directory,
            text=True,
            capture_output=True,
            check=True,
        )
        print(result.stdout)
```

Each child begins with a clean cache and a different starting location. Record both origins. In a real project, installation and distinctive package names remove this ambiguity; adding another guessed search directory would increase it.

Capture a useful import report

PYTHON
```
import importlib.util
import sys
from pathlib import Path


def import_report(name):
    """Return interpreter, working directory, and discovered origin for a name."""
    specification = importlib.util.find_spec(name)
    return {
        "name": name,
        "interpreter": sys.executable,
        "working_directory": str(Path.cwd()),
        "origin": specification.origin if specification else None,
    }


print(import_report("json"))
```

This diagnostic observes configuration without changing it. Preserve the exact command alongside the report because script mode and `-m` mode establish different entry context.

5. `__name__` separates import from direct execution

Every module receives a `__name__`. When imported normally, it is the import name. When Python executes a source file as the top-level program, it is `"__main__"`.

```python
# constellation_report.py
def build_report(labels):
    """Return one display line for constellation labels."""
    cleaned = [label.strip().title() for label in labels]
    return "Constellations: " + ", ".join(cleaned)


def main():
    """Run the small demonstration program."""
    print(build_report([" lyra", "orion "]))


if __name__ == "__main__":
    main()
```

The `if` line does not make `main` special. It simply compares a module-provided name with a string. During import, Python defines the functions but skips the call. During direct execution, the condition is true and the call runs.

Run both modes from the file's directory:

```bash
python constellation_report.py
python -c "import constellation_report; print(constellation_report.build_report(['vega']))"
```

The first prints the demo once. The second prints only the expression explicitly requested by the importer.

Keep the guarded block tiny. Put behavior in a callable so other Python code can reuse it and later tests can call it without launching a process. Unit 12 will develop full argument parsing, streams, and exit status; here, `main` only marks the execution boundary.

6. `python -m` executes a discovered module

The command:

```bash
python -m constellation_report
```

asks the selected interpreter to find `constellation_report` through the import system and execute it as `__main__`. This differs from passing a filesystem path such as `python tools/constellation_report.py`. Module mode works with the importable project structure and later supports packages through a `package/__main__.py` file.

Use `python -m pip` for the same identity reason: it asks the displayed Python interpreter to run its importable `pip` module. A bare `pip` command may be a launcher associated with another installation.

Compare the two launch contracts explicitly

From a directory containing `constellation_report.py`, run these diagnostics:

```python
# Add temporarily inside constellation_report.py while investigating.
import sys
from pathlib import Path

print("name:", __name__)
print("working directory:", Path.cwd())
print("first search entry:", sys.path[0])
```

Then compare:

```bash
python constellation_report.py
python -m constellation_report
```

Both execute the module as `__main__`, but the first names a source path and the second asks the import system to locate a module name. Depending on the command and platform, the first search entry can reflect the script directory or the current directory. Do not build application behavior around the incidental text of that entry. Use the commands to understand why one undocumented launch location can mask an import problem.

Write a launch contract in the README when a project gains a supported command:

```text
Working directory: project root
Interpreter:       project environment Python
Command:           python -m constellation_report
Input:             built-in demonstration observations
Output:            report text on standard output
```

This record makes reruns comparable. “I clicked Run in my editor” omits the interpreter, working directory, and launch mode, all of which can affect module discovery.

After the comparison, remove diagnostic prints from reusable source. Keep a small `main` and use an external command or logging/debugging tool when future investigation needs the same evidence. Importers should not inherit permanent diagnostic noise.

Editor Run buttons encode these same choices through settings: selected interpreter, launch file or module, working directory, arguments, and environment variables. When terminal and editor behavior differ, display that configuration rather than claiming editors use a different kind of Python. Reproduce the editor command in a terminal with the same interpreter and working directory. If it then matches, the source is probably not the differing variable. If it does not, compare the exact environment and arguments next.

Keep the launch record beside the project while diagnosing, then turn the supported route into concise README instructions. Debug-only variations such as running an internal module path should not become accidental public commands. One documented route, one fresh-process verification, and one callable action owner give future learners a stable place to begin.

Do not combine `-m` with a `.py` path

`python -m constellation_report` takes a dotted import name without `.py` or path separators. `python constellation_report.py` takes a filesystem path. They are two different command forms. :::

7. Build the multi-file constellation report

Create this structure in a disposable local folder:

```text
constellation-report/
├── orbit_math.py
├── report_text.py
└── run_report.py
```

Use these contracts:

```python
# orbit_math.py
def normalize_degrees(value):
    """Return an angle in the range 0 <= result < 360."""
    return value % 360
```

```python
# report_text.py
def observation_line(label, angle):
    """Return one normalized constellation observation line."""
    return f"{label.strip().title()}: {angle}°"
```

Complete the program entry module:

```python
# run_report.py
import orbit_math
import report_text


def build_report(observations):
    """Return newline-separated lines for `(label, angle)` observations."""
    lines = []
    for label, raw_angle in observations:
        angle = orbit_math.normalize_degrees(raw_angle)
        lines.append(report_text.observation_line(label, angle))
    return "\n".join(lines)


def main():
    observations = [(" lyra ", -15), ("orion", 390)]
    print(build_report(observations))


if __name__ == "__main__":
    main()
```

From the project folder, verify both use cases:

```bash
python run_report.py
python -c "import run_report; print(run_report.build_report([('vega', 720)]))"
```

Expected direct output:

```text
Lyra: 345°
Orion: 30°
```

The import command should print only `Vega: 0°`, with no demo report before it.

Make these controlled changes:

1. add a suffix parameter to `observation_line` without moving angle arithmetic into the text module;
2. temporarily add a top-level print to `orbit_math`, predict both commands' output, then remove it;
3. run from the parent directory and record the failure before returning to the documented working directory; and
4. print `orbit_math.__file__` to prove which source was loaded.

Compare a version with an explicit display suffix

PYTHON
```
# report_text.py
def observation_line(label, angle, suffix="°"):
    """Return one normalized constellation observation line."""
    return f"{label.strip().title()}: {angle}{suffix}"
```

PYTHON
```
# run_report.py
import orbit_math
import report_text


def build_report(observations, suffix="°"):
    """Return newline-separated lines for `(label, angle)` observations."""
    lines = []
    for label, raw_angle in observations:
        angle = orbit_math.normalize_degrees(raw_angle)
        lines.append(report_text.observation_line(label, angle, suffix))
    return "\n".join(lines)
```

Angle normalization remains in `orbit_math`; display policy remains in `report_text`; orchestration remains in `run_report`. The files are separated by responsibility rather than by arbitrary line count.

Checkpoint: choose a dependable execution boundary

Diagnose search and entry-point behavior

Choose the repair or explanation that matches the evidence.

1. An import works in one terminal but not another. Which evidence set is most useful first?

- **A.** Only the source file's line count
- **B.** Only the computer model
- **C.** A list of random packages to install
- **D.** Exact command, working directory, `sys.executable`, search path, and discovered origin

2. Why place `main()` under `if __name__ == "__main__"`?

- **A.** Importers receive definitions without automatically running the program action.
- **B.** It makes all functions private.
- **C.** It installs the module.
- **D.** It forces every import to reload.

3. Which command correctly asks Python to execute an importable module named `run_report`?

- **A.** `python -m run_report.py`
- **B.** `python -m run_report`
- **C.** `python import run_report`
- **D.** `python --module ./run_report.py`

Write your choices before checking the answer key at the end of this book.

8. Key points for modules and execution

- A module is initialized by executing its top-level code in its own namespace.
- `import module` keeps ownership visible; selected imports bind individual objects, and star imports obscure what entered the caller.
- Successfully initialized modules are cached in `sys.modules`. Use a fresh process when stale notebook state makes an experiment ambiguous.
- Import names are discovered through configured finders and search locations; they are not arbitrary relative filesystem paths.
- Diagnose failures with the exact command, working directory, interpreter, search path, and module origin before changing anything.
- Keep import-time work limited to establishing the module. Put program action in a callable and guard the call with `if __name__ == "__main__"`.
- `python file.py` executes a path; `python -m dotted.name` discovers and executes a module with the chosen interpreter.

References

- Python tutorial: modules
- Python language reference: the import system
- `sys.path`, `sys.modules`, and `sys.executable`
- `importlib` implementation and inspection utilities
- `__main__` and top-level code environments

Design a Package People Can Rely On

Online lesson and runnable notebook

Organize related modules as a package, design stable public imports, prevent import-time surprises and cycles, and move implementation without breaking callers.

- **Level:** Python Foundations
- **Estimated time:** 4–5 hours
- **You will learn:** Create a real import package, use package-relative imports, define a small public facade, recognize import side effects and cycles, and preserve callers while implementation files change.
- **Practice in:** A local editor and terminal using a disposable package workspace

The constellation report now has several related modules. Shipping `orbit_math.py`, `report_text.py`, and `run_report.py` as unrelated top-level names risks collisions with somebody else's files and exposes every internal location to callers. A package gives the project one import namespace. A carefully chosen package interface gives callers a smaller promise than "every name in every file will remain here forever."

This lesson asks:

- What turns a directory of source files into the package used here?
- When should an internal import be absolute or relative?
- Which imports do callers deserve to treat as stable?
- What do a leading underscore and `__all__` communicate—and not enforce?
- How do import-time work and circular dependencies make packages fragile?

1. Put related modules under one package name

Create this local structure:

```text
constellation-workspace/
├── constellation/
│   ├── __init__.py
│   ├── angles.py
│   └── reports.py
└── demo.py
```

For this course, a package is a directory containing `__init__.py`. Python also supports namespace packages without that file, but their multi-location design is outside this foundation lesson. An explicit `__init__.py` gives this small project one clear initialization file.

```python
# constellation/angles.py
FULL_CIRCLE = 360
```

PYTHON (CONTINUED)

```
def normalize_degrees(value):
    """Return an angle in the range 0 <= result < 360."""
    return value % FULL_CIRCLE
```

PYTHON

```
# constellation/reports.py
from .angles import normalize_degrees


def observation_line(label, raw_angle):
    """Return one display line with a normalized angle."""
    angle = normalize_degrees(raw_angle)
    return f"{label.strip().title()}: {angle}°"
```

The leading dot in `.angles` means “start in the current package.” It tells a reader that `angles` is a sibling owned by `constellation`, not an unrelated top-level package.

Expose the first public names from the package root:

PYTHON

```
# constellation/__init__.py
from .angles import normalize_degrees
from .reports import observation_line

__all__ = ["normalize_degrees", "observation_line"]
```

Now the caller uses the package facade:

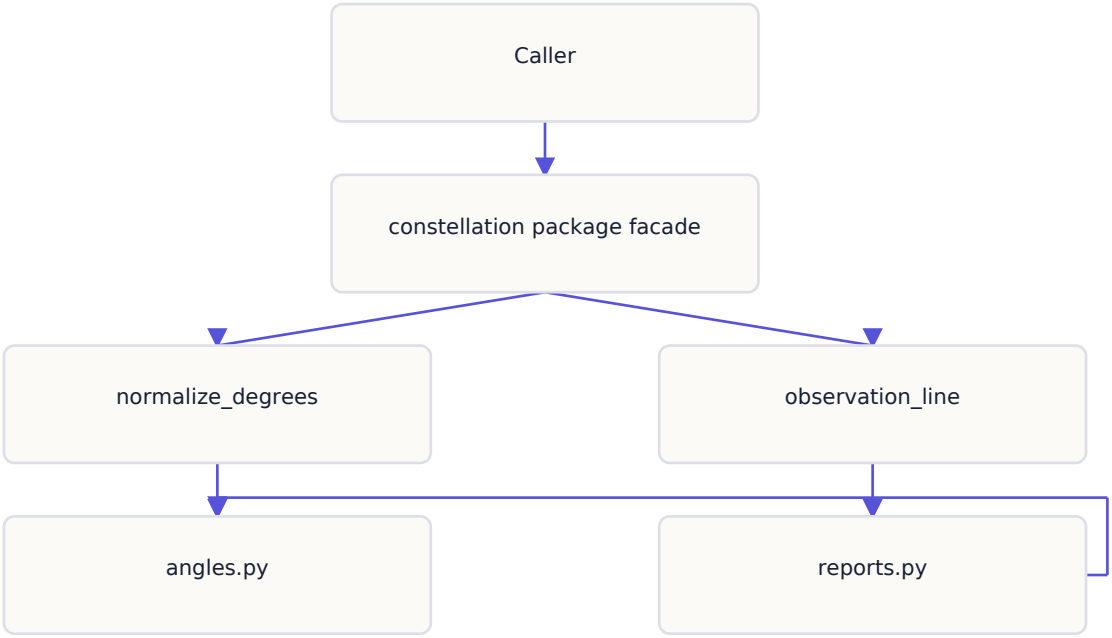
PYTHON

```
# demo.py
from constellation import normalize_degrees, observation_line

print(normalize_degrees(-15))
print(observation_line(" lyra ", 390))
```

From `constellation-workspace`, run `python demo.py`. The caller does not need to know that normalization currently lives in `angles.py`.

The package root is a front desk. Callers use the stable public names while the front desk delegates to internal modules.



What Python initializes

When Python imports `constellation.reports`, it first establishes the `constellation` package and runs `constellation/__init__.py`, then initializes the requested submodule as needed. Therefore the package initializer should be lightweight. Re-exporting selected definitions is normal; reading user input or generating a report is not.

Checkpoint: read the package tree

Connect package files and import names

Choose the statement supported by the package structure.

1. Inside `constellation/reports.py`, what does `from .angles import normalize_degrees` mean?

- **A.** Search only third-party distributions.
- **B.** Import a file from the working directory regardless of package.
- **C.** Import `angles` relative to the current `constellation` package.
- **D.** Move the function into `reports.py`.

2. Why should `constellation/__init__.py` avoid prompts and report generation?

- **A.** Python never executes `__init__.py`.
- **B.** Only private functions may appear there.
- **C.** It is not valid Python.
- **D.** Importing the package or a submodule can initialize it, so unrelated effects surprise callers.

3. What does re-exporting `normalize_degrees` from the package root allow?

- **A.** Callers can use a stable `from constellation import normalize_degrees` path while internals evolve.

- **B.** The function runs without being called.
- **C.** No internal modules are needed.
- **D.** The name becomes inaccessible from Python.

Write your choices before checking the answer key at the end of this book.

2. Use absolute imports for package identity and relative imports for nearby ownership

Within a package, these two statements may reach the same module:

```python
# Absolute: begins from an importable top-level package.
from constellation.angles import normalize_degrees

# Explicit relative: begins from the current package.
from .angles import normalize_degrees
```

The first spells the full package identity and is clear across distant package areas. The second emphasizes that the dependency is a nearby sibling. Choose a consistent policy rather than mixing forms randomly.

Relative imports depend on package context. Do not run a package's internal file as though it were an independent script:

```bash
# Fragile command: reports.py has no parent-package context here.
python constellation/reports.py
```

Instead, run a top-level program that imports the package, or provide a package entry module and use module mode:

```bash
python demo.py
# Later: python -m constellation
```

The internal module is reusable implementation, not a command merely because it ends in `.py`.

Import modules rather than duplicating definitions

Do not copy `normalize_degrees` into `reports.py` to avoid an import. Two copies can drift into different contracts. Keep one owner and express the dependency:

```python
# constellation/reports.py
from . import angles


def observation_line(label, raw_angle):
    angle = angles.normalize_degrees(raw_angle)
    return f"{label.strip().title()}: {angle}°"
```

Importing the module rather than the selected function can make ownership more visible. Either form is valid; choose for clarity and cycle structure, not a claim that one spelling is universally faster.

3. Design a public facade deliberately

A **public interface** is the set of names callers are invited to rely on. It is a design and compatibility promise, not every technically reachable object.

Use three signals together:

1. Documentation teaches the supported imports and behavior.
2. Package-root re-exports offer short, stable paths.
3. Naming marks implementation details with a leading underscore.

```python
# constellation/reports.py
def _clean_label(value):
    """Normalize an internal display label."""
    return value.strip().title()


def observation_line(label, raw_angle):
    """Return one supported report line."""
    return f"{_clean_label(label)}: {raw_angle % 360}°"
```

The underscore does not create privacy or access control. A determined caller can still import `_clean_label`. It communicates “this name may change; prefer the documented public operation.”

`__all__` also communicates intent and controls what a star import exports:

```python
# constellation/__init__.py
from .angles import normalize_degrees
from .reports import observation_line

__all__ = ["normalize_degrees", "observation_line"]
```

It does not block access to other attributes, and it does not replace documentation. Since application code should avoid star imports, its greater value here is an inspectable list of intended exports.

Check the facade explicitly:

```python
import constellation

print(constellation.__all__)
print(callable(constellation.normalize_degrees))
print(callable(constellation.observation_line))
```

Keep the public surface smaller than the implementation

Do not automatically re-export every helper. Each public name becomes a future compatibility decision. A focused package root helps beginners and tools find the supported path:

```python
from constellation import observation_line

print(observation_line("andromeda", 725))
```

Callers can still import a documented submodule when that submodule itself is a public part of a larger package. “Everything must be at the root” is no better than “nothing should be at the root.” Design around coherent user tasks.

Checkpoint: identify the actual promise

Separate public promises from implementation details

Choose the interpretation a caller should use.

1. What does the leading underscore in `_clean_label` normally communicate?

- **A.** Python encrypts the function.
- **B.** It is an implementation detail rather than a supported public import.
- **C.** The function can only return numbers.
- **D.** The function runs only during installation.

2. Which claim about `__all__` is accurate?

- **A.** It prevents all access to omitted names.
- **B.** It installs the package.
- **C.** It lists intended star-import exports but does not enforce privacy.
- **D.** It automatically documents every contract.

3. Why avoid re-exporting every internal helper from the package root?

- **A.** Python packages may export only two names.
- **B.** Imports stop working after three exports.
- **C.** Helpers cannot have docstrings.
- **D.** Every advertised name enlarges what callers may expect to remain stable.

Write your choices before checking the answer key at the end of this book.

4. Keep imports quiet and predictable

Importing a package should establish definitions and lightweight constants. It should not unexpectedly:

- prompt for user input;
- print a demonstration;
- create or replace files;
- connect to a remote service;
- start a long computation; or
- launch the command-line program.

This initializer is surprising:

```python
# Avoid in constellation/__init__.py
from .reports import observation_line

print(observation_line("demo star", 0))
```

Move the program action to a callable and an execution boundary. A package can support `python -m constellation` through `__main__.py`:

```python
# constellation/__main__.py
from .reports import observation_line


def main():
    print(observation_line("demo star", 0))


if __name__ == "__main__":
    main()
```

Now `import constellation` is quiet, while `python -m constellation` asks for the action. An installed console command is another entry route taught in Lesson 6. All routes should eventually delegate to one callable rather than copying program logic.

Avoid mutable package state as hidden communication

This pattern couples every caller to import order:

```python
# Avoid as a project communication mechanism.
active_observatory = None
```

One module sets it; another hopes it has already been set. Prefer passing configuration or data explicitly to functions. Module constants are appropriate for fixed definitions such as `FULL_CIRCLE = 360`; changing session state is a different ownership problem.

Verify import behavior in a fresh process

A notebook that imported yesterday's package object cannot prove today's source is quiet. Run a new interpreter and capture all three process channels:

```python
import subprocess
import sys

result = subprocess.run(
    [
        sys.executable,
        "-c",
        "import constellation; "
        "print(sorted(constellation.__all__))",
    ],
    text=True,
    capture_output=True,
)

print("return code:", result.returncode)
print("stdout lines:", result.stdout.splitlines())
print("stderr:", result.stderr)
```

For the package in this lesson, the return code should be zero, stderr should be empty, and stdout should contain only the one line explicitly printed by the diagnostic. A demo line appearing before it proves import-time action remains. An exception traceback on stderr identifies initialization that did not finish.

Then ask for each public operation through the advertised path:

```python
program = (
    "from constellation import normalize_degrees, observation_line; "
    "print(normalize_degrees(-1)); "
    "print(observation_line(' vega ', 361))"
)
result = subprocess.run(
    [sys.executable, "-c", program],
    text=True,
    capture_output=True,
    check=True,
)
assert result.stdout.splitlines() == ["359", "Vega: 1°"]
```

This contract cares about supported imports and behavior, not whether the implementation uses a particular helper name. That distinction lets a refactor remain invisible to callers.

Treat the package facade as a dependency direction

It is tempting to make internal modules import names back from the root:

```python
# Avoid inside constellation/reports.py:
# from constellation import normalize_degrees
```

The package root is still initializing and imports `reports.py`; `reports.py` then asks the unfinished root for a re-export. Instead, internal modules import their actual lower-level owner (`.angles` or `.calculations`). The root facade depends on internals, not the other way around.

Annotate the package before refactoring:

File	Defines	Imports from	Re-exported at root?
`calculations.py`	`normalize_degrees`	nothing internal	yes
`reports.py`	`observation_line`	`calculations.py`	yes
`__init__.py`	public facade	both modules	not applicable
`__main__.py`	execution boundary	command/report layer	no

If an arrow points upward from calculations to reports or from an internal module to the facade, ask which responsibility is misplaced. The table is more useful than moving an import into a function solely to delay the same cycle.

5. Break circular dependencies by moving shared direction downward

A circular import occurs when initialization returns to a module that has not finished defining the needed names. Consider this dependency:

```text
reports.py  ->  labels.py
    ^              |
    |______________|
```

`reports.py` imports `clean_label` from `labels.py`, while `labels.py` imports `observation_line` from `reports.py`. Whichever starts first encounters a partially initialized partner.

The repair is usually architectural, not “put imports randomly inside functions.” Find the shared lower-level responsibility:

```text
labels.py       angles.py
     \           /
      \         /
       reports.py
```

```python
# constellation/labels.py
def clean_label(value):
    """Return a trimmed display label."""
    return value.strip().title()
```

```python
# constellation/reports.py
from .angles import normalize_degrees
from .labels import clean_label


def observation_line(label, raw_angle):
    """Return one display line using lower-level helpers."""
    return f"{clean_label(label)}: {normalize_degrees(raw_angle)}°"
```

Both helpers point toward report composition; neither needs to import the higher-level report. This one-directional dependency is easier to initialize, understand, and change.

Read the earliest useful traceback line

Messages about a “partially initialized module” or a name missing during import often indicate a cycle. Draw module-to-module arrows from the import statements, then move genuinely shared behavior to a lower-level owner. Renaming imports or retrying usually does not remove the cycle. :::

6. Import-package names and distribution names serve different users

The source directory here is named `constellation`, so callers write `import constellation`. In Lesson 6, a distribution might be named `constellation-report` in `pyproject.toml`, because distribution names often use hyphens and describe the installable project.

```text
Distribution metadata: constellation-report 0.1.0
Import package:         constellation
Module:                 constellation.reports
Public callable:        constellation.observation_line
```

These are related identities, not interchangeable spellings. One distribution can install multiple import packages; an import package can contain many modules. Write installation and metadata instructions with the distribution name, and Python examples with the import name.

7. Move an implementation file without breaking callers

Start with the package from Section 1. The public contract is:

PYTHON
```
from constellation import normalize_degrees, observation_line

assert normalize_degrees(-1) == 359
assert observation_line(" vega ", 361) == "Vega: 1°"
```

Now replace `angles.py` with an internal `calculations.py`:

TEXT
```
constellation/
├── __init__.py
├── calculations.py
└── reports.py
```

PYTHON
```
# constellation/calculations.py
FULL_CIRCLE = 360


def normalize_degrees(value):
    """Return an angle in the range 0 <= result < 360."""
    return value % FULL_CIRCLE
```

Update internal imports and keep the root facade:

PYTHON
```
# constellation/reports.py
from .calculations import normalize_degrees


def observation_line(label, raw_angle):
    """Return one normalized constellation observation line."""
    angle = normalize_degrees(raw_angle)
    return f"{label.strip().title()}: {angle}°"
```

PYTHON
```
# constellation/__init__.py
from .calculations import normalize_degrees
from .reports import observation_line

__all__ = ["normalize_degrees", "observation_line"]
```

Rerun the public contract unchanged. A caller that used `from constellation.angles import normalize_degrees` would break because it coupled itself to the old implementation file. A caller using the advertised facade remains valid.

Next, add `format_observations(observations)` to `reports.py`. It should return newline-separated `observation_line` results. Export it only after its name, input shape, return type, and empty-input behavior are deliberate.

Compare one package extension after making your own

PYTHON
```
# constellation/reports.py
from .calculations import normalize_degrees


def observation_line(label, raw_angle):
    """Return one normalized constellation observation line."""
    angle = normalize_degrees(raw_angle)
    return f"{label.strip().title()}: {angle}°"


def format_observations(observations):
    """Return newline-separated lines for `(label, angle)` pairs."""
    return "\n".join(
        observation_line(label, angle)
        for label, angle in observations
    )
```

PYTHON
```
# constellation/__init__.py
from .calculations import normalize_degrees
from .reports import format_observations, observation_line

__all__ = [
    "format_observations",
    "normalize_degrees",
    "observation_line",
]
```

PYTHON
```
from constellation import format_observations

assert format_observations([]) == ""
assert format_observations([("lyra", -1), ("orion", 360)]) == (
    "Lyra: 359°\nOrion: 0°"
)
```

The public name is added in one visible place. Its implementation still depends downward on calculations rather than creating a reverse import.

Checkpoint: keep the package dependable while it changes

Repair import structure without breaking callers

Choose the design that preserves a clear dependency direction.

1. `normalize_degrees` **moves from** `angles.py` **to** `calculations.py`**. Which caller is protected by the updated facade?**

- **A.** `from constellation import normalize_degrees`
- **B.** `from constellation.angles import normalize_degrees`
- **C.** `python constellation/angles.py`
- **D.** A caller that copied the old file

2. Two modules import higher-level behavior from each other. What is the strongest first redesign?

- **A.** Add more star imports.

- **B.** Extract genuinely shared lower-level behavior so dependencies point one way.
- **C.** Catch `ImportError` and continue.
- **D.** Mutate `sys.path` during import.

3. Where should package demo action live if import must remain quiet but `python -m constellation` should run it?

- **A.** At top level in `__init__.py`
- **B.** Inside every helper module
- **C.** In a callable invoked by `constellation/__main__.py`
- **D.** In `__all__`

Write your choices before checking the answer key at the end of this book.

Review the facade as a future caller

Before declaring the package finished, open a fresh Python process and answer these questions without reading internal files first:

1. Can a caller discover the supported operations from package documentation and `__all__`?
2. Do the root names use task language rather than internal file names?
3. Does importing the root or any advertised submodule remain quiet?
4. Do public functions keep their contract when implementation modules move?
5. Does `python -m constellation` delegate to one command owner rather than duplicating package logic?

Create a caller-only check:

PYTHON

```
import subprocess
import sys
from tempfile import TemporaryDirectory

with TemporaryDirectory() as outside_directory:
    result = subprocess.run(
        [
            sys.executable,
            "-c",
            "import constellation; "
            "assert set(constellation.__all__) == "
            "{'normalize_degrees', 'observation_line'}; "
            "print(constellation.observation_line('lyra', -15))",
        ],
        cwd=outside_directory,
        text=True,
        capture_output=True,
        check=True,
    )
    assert result.stdout == "Lyra: 345°\n"
    assert result.stderr == ""
```

This proof assumes the package is installed in the selected interpreter, which Lessons 5–6 make explicit. The unrelated working directory prevents a local `constellation/` folder from becoming the accidental provider.

Now imagine changing `_clean_label` to two helpers. The caller check should not change. If it does, an implementation detail escaped into the advertised contract. Conversely, adding a genuinely supported formatter should begin with its caller example, input/return/failure decisions, and root export—not with an automatic re-export of every new definition.

Review error behavior too. A facade should not catch every internal exception merely to look stable. Preserve meaningful contract errors, add context only where the package understands it, and avoid exposing an internal filename as the only explanation users receive. Stability covers documented behavior and failure expectations, not just whether an import statement still parses. Run the caller proof after every internal move, not only before release.

8. Key points for dependable packages

- A package groups related modules under one import namespace; this course uses an explicit `__init__.py` for a clear beginner structure.
- Explicit relative imports show nearby package ownership. Absolute imports show the full top-level package identity.
- Package initialization happens during imports, so keep `__init__.py` lightweight and free of unrelated action.
- A facade re-exports selected supported names. Leading underscores and `__all__` communicate intent but do not enforce privacy.
- Public interfaces should be useful and small enough to support deliberately.
- Repair import cycles by clarifying responsibility and dependency direction, not by scattering delayed imports without understanding the design.
- Distribution, import-package, module, and callable names are related but distinct identities.
- A stable root import lets implementation files move without forcing every caller to change.

References

- Python tutorial: packages
- Python import reference: regular packages
- Python tutorial: intra-package references
- Python language reference: `__all__`
- PyPA explanation of distribution packages and import packages

Build a Project You Can Navigate and Share

Online lesson and runnable notebook

Arrange a clear src-layout project, remove working-directory assumptions, ignore generated files, and record understandable changes with essential Git commands.

- **Level:** Python Foundations
- **Estimated time:** 4–5 hours
- **You will learn:** Give source, checks, documentation, configuration, and generated output predictable homes; explain why a `src` layout requires installation; and inspect, stage, commit, ignore, and branch in a disposable Git repository.
- **Practice in:** A local editor and terminal using a disposable project

Imagine opening a collaborator’s folder and finding Python files, virtual environments, exported reports, notes, and old copies mixed together. Which file starts the program? Which code may callers import? Which output can be deleted? Which changes will the next Git commit contain? A project layout answers those questions before a person has to open every file.

You will arrange the constellation package and record three small snapshots. Keep these questions nearby:

- What responsibility does each top-level path communicate?
- Why does `src/` deliberately prevent an uninstalled import from the repository root?
- How can commands avoid relying on an accidental working directory?
- What is the difference between working-tree, staged, and committed content?
- Which files are source, and which are reproducible local artifacts?

1. Give every project responsibility a visible home

Build toward this tree:

```text
constellation-report/
├── .gitignore
├── README.md
├── pyproject.toml
├── src/
│   └── constellation/
│       ├── __init__.py
│       ├── __main__.py
│       ├── angles.py
│       └── reports.py
└── checks/
    └── check_public_api.py
```

Each part answers a different question:

Path	Responsibility
`pyproject.toml`	standardized build and project metadata plus tool configuration
`README.md`	human-facing purpose and first-use instructions
`src/constellation/`	importable product source
`checks/`	executable assertions that verify visible behavior
`.gitignore`	local/generated paths Git should normally leave untracked
`.git/`	Git's private repository database, created by `git init`

Do not create folders merely because a template has them. Add `docs/`, `data/`, or `notebooks/` when the project truly owns those responsibilities. A clear small tree is better than a large empty architecture.

Create the starting directories from the project root:

BASH
```
mkdir -p src/constellation checks
touch src/constellation/__init__.py
```

On Windows PowerShell, use `New-Item -ItemType Directory` and `New-Item` or create the paths in the editor. The tree is the contract; shell spelling can vary by platform.

Source belongs under one import-package directory

PYTHON
```
# src/constellation/angles.py
FULL_CIRCLE = 360


def normalize_degrees(value):
    """Return an angle in the range 0 <= result < 360."""
    return value % FULL_CIRCLE
```

PYTHON
```
# src/constellation/reports.py
from .angles import normalize_degrees

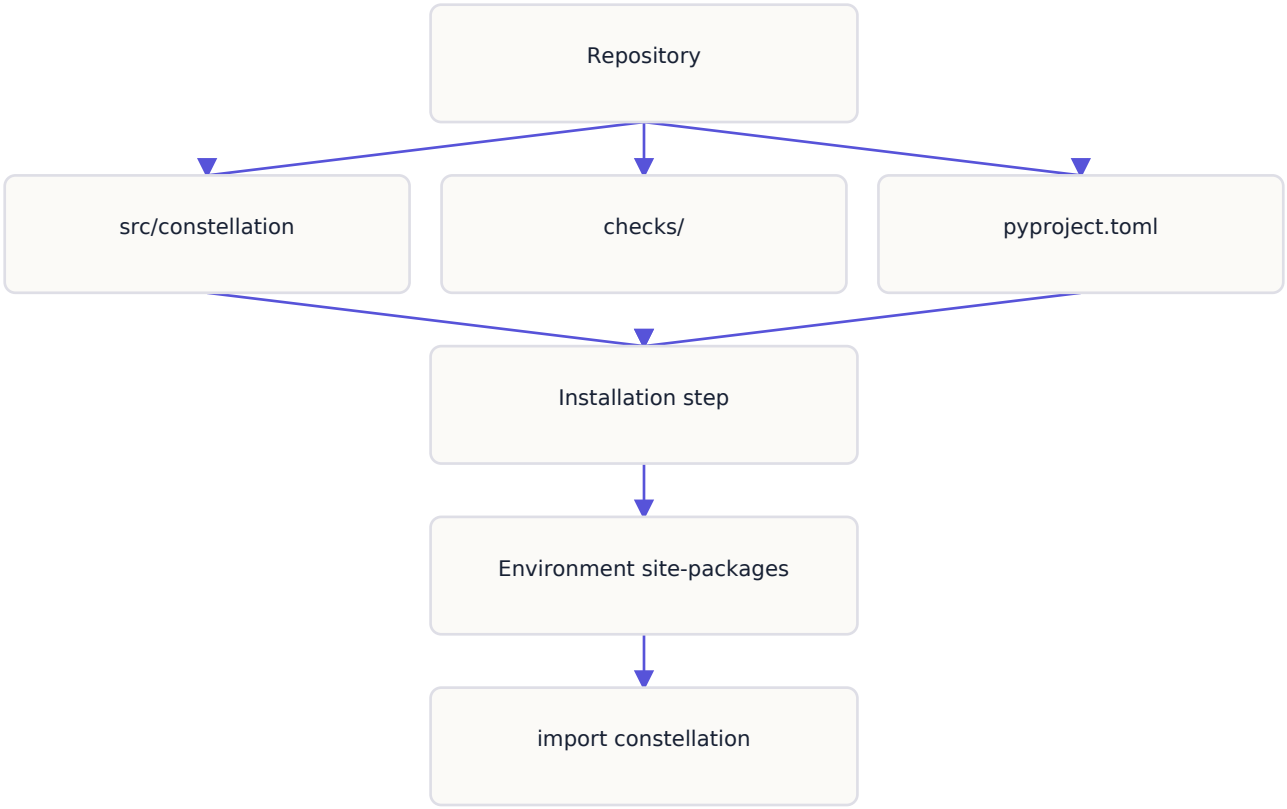

def observation_line(label, raw_angle):
    """Return one normalized constellation line."""
    angle = normalize_degrees(raw_angle)
    return f"{label.strip().title()}: {angle}°"
```

PYTHON
```
# src/constellation/__init__.py
from .angles import normalize_degrees
from .reports import observation_line

__all__ = ["normalize_degrees", "observation_line"]
```

`src` is not part of the import name. After installation, callers use `import constellation`, not `import src.constellation`.

The repository contains source and project support files. Installation makes the package importable in an environment; callers do not import through the repository folder name.



Checkpoint: classify the tree

Give each file one understandable role

Choose the path that owns the stated responsibility.

1. Where should standardized project and build metadata live?

- **A.** Inside `.git/`
- **B.** In a generated wheel only
- **C.** In every source module
- **D.** In `pyproject.toml` at the project root

2. After the src-layout project is installed, what should a caller import?

- **A.** `constellation`
- **B.** `src.constellation`
- **C.** `constellation-report.src`
- **D.** The repository directory

3. Why keep `checks/check_public_api.py` outside the import package?

- **A.** Checks are invalid Python.

- **B.** It separates verification support from code shipped as the product package.
- **C.** Python cannot open nested files.
- **D.** Git tracks only top-level directories.

Write your choices before checking the answer key at the end of this book.

2. Let the src layout expose accidental imports

With package source directly at the repository root, launching Python there can make that source importable even if the project was never installed. A `src` layout removes that convenient coincidence.

Before installation, this command from `constellation-report/` should fail:

```bash
python -c "import constellation"
```

That failure is useful. It prevents a check from passing against source files that a built distribution might forget to include. Lesson 6 will add an editable installation during development and a clean wheel installation for final proof.

Do **not** repair the failure this way:

```python
# Avoid in project source.
import sys

sys.path.insert(0, "src")
```

The hard-coded path depends on the current working directory and lets every entry script invent a different import configuration. Install the project into the intended environment instead.

Keep resource and configuration roots explicit

Avoid guessing a repository root by walking a fixed number of parents from `__file__`:

```python
# Brittle: package depth and installation layout are not configuration contracts.
from pathlib import Path

guessed_repository = Path(__file__).resolve().parents[2]
```

After wheel installation, that path points inside the environment, not back to the user's source checkout. Choose among these contracts instead:

- caller-supplied input/output paths;
- current-working-directory paths when the command explicitly documents that interface;
- installed package resources accessed with `importlib.resources`; or
- configuration supplied through the later command/configuration layer.

For a function that writes a report, pass the destination:

```python
from pathlib import Path
```

PYTHON (CONTINUED)

```
def save_report(text, destination):
    """Write report text to an explicit destination and return that path."""
    path = Path(destination)
    path.write_text(text, encoding="utf-8")
    return path
```

The caller owns location policy. The package owns how to write the text.

Verify behavior from a temporary destination

PYTHON

```
from pathlib import Path
from tempfile import TemporaryDirectory

with TemporaryDirectory() as temporary_name:
    output = Path(temporary_name) / "night-report.txt"
    saved = save_report("Vega: 1°\n", output)
    assert saved == output
    assert output.read_text(encoding="utf-8") == "Vega: 1°\n"
```

This check supplies its dependency and cleans up its artifact. It works no matter where the notebook itself started.

3. Ignore artifacts that can be recreated locally

Python, environments, build tools, editors, and operating systems create local files. Record project source and deliberate configuration; normally ignore reproducible caches and outputs:

GITIGNORE

```
# Virtual environments
.venv/

# Python caches and installed-project metadata
__pycache__/
*.py[cod]
*.egg-info/

# Build outputs
build/
dist/

# Tool caches
.pytest_cache/
.mypy_cache/
.ruff_cache/

# Local editor and operating-system files
.DS_Store
```

Do not blindly ignore all data, configuration, notebooks, or lock files. Ask whether the repository deliberately owns the file and whether collaborators need the same content. A generated report normally differs from a hand-curated example fixture.

After adding `.gitignore`, inspect ignored paths:

BASH

```
git status --short --ignored
git check-ignore -v .venv/ dist/example.whl
```

`git check-ignore -v` reports the exact ignore rule and file that matched. If a file was already tracked, adding it to `.gitignore` does not erase it from history or automatically stop tracking it. Ignore rules primarily affect untracked paths.

Checkpoint: remove hidden location assumptions

Choose portable project boundaries

Choose the design that survives installation and a changed working directory.

1. Why is `import constellation` failing from the repository root before installation useful in a src layout?

- **A.** It proves the source has syntax errors.
- **B.** It means `src` must enter the import name.
- **C.** It exposes that importability should come from installation rather than the repository's current directory.
- **D.** It prevents the project from ever being packaged.

2. Which design best avoids guessing where a report should be written?

- **A.** Always write beside `__file__`.
- **B.** Walk exactly two parent directories.
- **C.** Search the entire disk for the repository.
- **D.** Accept the destination path as a function or command input.

3. A generated file is already tracked. What does adding its pattern to `.gitignore` do by itself?

- **A.** It does not automatically remove the tracked file from history or tracking.
- **B.** It deletes every matching file.
- **C.** It rewrites prior commits.
- **D.** It moves the file into `src/`.

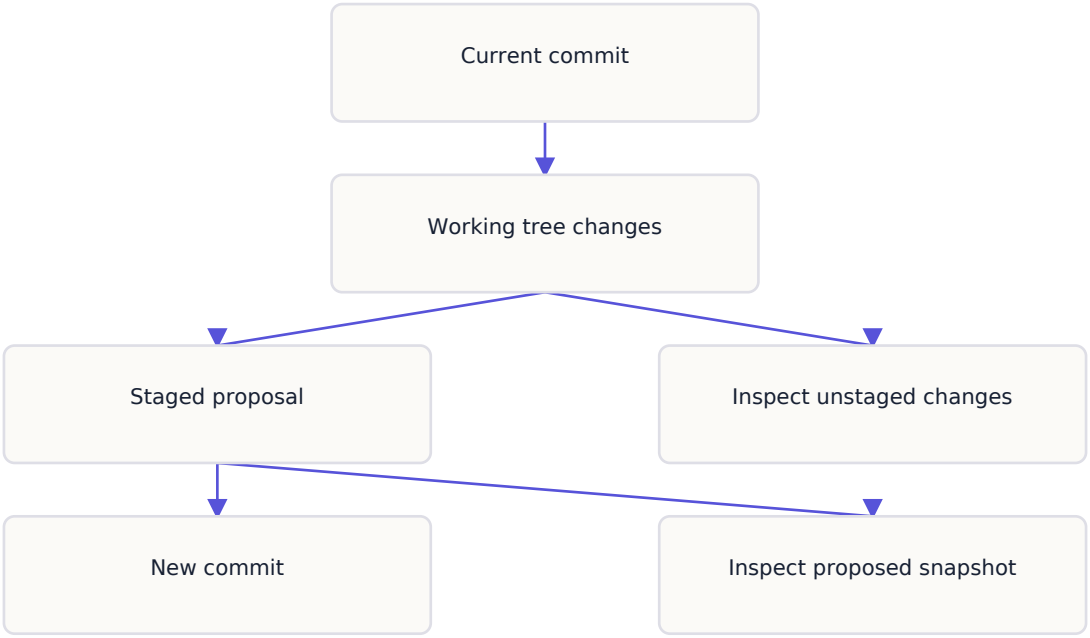
Write your choices before checking the answer key at the end of this book.

4. Git records snapshots, not a cloud folder

Git is a version-control system. A repository stores a graph of snapshots and the relationships between them. Three everyday views matter first:

1. the **working tree** contains files you are currently editing;
2. the **staging area** records the exact content proposed for the next commit;
3. the current **commit** is the checked-out recorded snapshot.

Editing, staging, and committing are separate actions. Inspection can happen at every boundary.



Current commit → Working tree changes: edit files

Working tree changes → Staged proposal: git add selected paths

Staged proposal → New commit: git commit

Working tree changes → Inspect unstaged changes: git diff

Staged proposal → Inspect proposed snapshot: git diff --staged

Initialize only inside the disposable `constellation-report` folder:

```bash
git init
git status
```

Before the first commit, configure an author identity if Git asks. You may set it for this repository rather than changing every repository:

```bash
git config user.name "Course Learner"
git config user.email "learner@example.invalid"
```

Use your real project identity outside a disposable exercise.

Inspect before staging

```bash
git status --short
git diff
```

For untracked files, `git diff` has no earlier tracked content to compare. Read the files and status list. Stage only the coherent first snapshot:

```bash
git add .gitignore README.md pyproject.toml src checks
git status --short
git diff --staged
```

In short status, the first column describes the index/staging state; the second describes the working tree. For example, `M` means staged modification, `M` means unstaged modification, and `??` means untracked.

Commit only after the staged diff contains what the message claims:

```bash
git commit -m "Create constellation package skeleton"
git log --oneline --decorate -n 3
```

A commit is local. It does not automatically upload anywhere. Remotes, hosting, and collaboration workflows build on these local concepts.

5. Stage a file twice when it changes twice

Staging copies the file's **current content** into the proposed snapshot. If you edit the working file again, the staged version and working version differ.

Try this controlled sequence:

```bash
printf "# Constellation Report\n" > README.md
git add README.md
printf "\nBuild night-sky observation reports.\n" >> README.md
git status --short
git diff
git diff --staged
```

The staged diff contains the title. The unstaged diff contains the later description relative to the staged version. Run `git add README.md` again only if both lines belong in the next commit.

This distinction prevents "I staged it earlier" from becoming accidental loss of later work. Always review `git diff --staged` immediately before committing.

Unstage a mistaken path without deleting the work

Suppose `git diff --staged` reveals that `notes/private-scratch.txt` entered the proposal accidentally. Unstage it while leaving the working file available:

```bash
git restore --staged notes/private-scratch.txt
git status --short
git diff --staged
```

`--staged` changes the index proposal. It does not mean "delete my working file." This is different from:

```bash
# Read before using: this discards unstaged working-tree edits to README.md.
git restore README.md
```

Never paste a restore command merely because status looks untidy. First decide which copy—current commit, staged proposal, or working edit—you intend to keep. When the work matters, copy it outside the experiment or make a deliberate snapshot before practicing destructive recovery.

For a new untracked file, `git restore` has no committed version to recover. Untracked work needs its own deletion or preservation decision. `git clean` can delete many untracked files and is intentionally outside this beginner lab.

Translate short status one path at a time

Use this sequence whenever the two columns are confusing:

```bash
git status --short
git diff -- README.md
git diff --staged -- README.md
```

Then fill a small table:

Question	Answer for `README.md`
Is a version tracked in the current commit?	inspect status/history
What content is staged?	inspect `git diff --staged -- README.md`
What later content is only in the working tree?	inspect `git diff -- README.md`
What should the next commit claim?	compare both diffs with the intended change

Path-limited diffs reduce noise without changing any state. Once the answer is clear, stage again, unstage, or keep editing. Inspection commands are safe to repeat.

Write commits around one understandable change

Useful beginner commits might be:

```
1  Create constellation package skeleton
2  Add normalized observation report
3  Document local verification command
```

Avoid one commit that mixes a feature, unrelated formatting, generated build artifacts, and experimental notes. Small coherent snapshots make history useful for explanation and recovery.

6. Branches let names point at different lines of work

A branch is a movable name for a commit, not a duplicate folder by definition. Create a short-lived branch before trying a report-format change:

```bash
git switch -c report-header
git branch --show-current
```

Edit and commit the focused change, then inspect the graph:

```bash
git log --oneline --decorate --graph --all -n 8
```

Switching branches updates tracked working-tree files to the selected snapshot. Git may refuse when uncommitted changes would be overwritten. Read that refusal instead of forcing it. Merging, resolving conflicts, rebasing, and remote branch collaboration deserve later guided practice; this lesson's boundary is creating, identifying, and inspecting a branch without losing work.

Read history as a project explanation

`git log --oneline` abbreviates identifiers and subjects for navigation. Ask Git for one complete snapshot when you need its actual change:

```bash
git show --stat HEAD
git show --format=fuller --no-ext-diff HEAD
```

The first summarizes affected paths. The second displays commit metadata and patch. A message such as `Update files` explains almost nothing; `Add normalized observation report` lets a reader predict what the patch should contain.

Commit identifiers are derived from commit content and relationships, not sequence numbers such as "commit 3." Two learners following the same lesson can receive different identifiers because author, timestamp, or content differs. Refer to an identifier or branch name when discussing a particular snapshot.

Before switching away from `report-header`, run:

```bash
git status --short
git log --oneline --decorate --graph --all -n 8
```

If status is not clean, decide whether the work belongs in a coherent commit or should remain unfinished. Do not assume branch switching stores arbitrary working changes as a new snapshot; commits create the history you can later explain.

7. Record a three-snapshot project story

Complete the constellation tree and create these snapshots:

Snapshot 1: the package skeleton

- `.gitignore`
- `README.md` with project purpose
- `src/constellation/__init__.py`
- placeholder `pyproject.toml` comment if Lesson 6 metadata is not written yet

Snapshot 2: observable behavior

Add `angles.py`, `reports.py`, and this built-in assertion script:

```python
# checks/check_public_api.py
from constellation import normalize_degrees, observation_line

assert normalize_degrees(-1) == 359
assert observation_line(" vega ", 361) == "Vega: 1°"
print("public API checks passed")
```

The import will work after Lesson 6 installs the project. Until then, record the expected failure rather than adding `src` to `sys.path`.

Snapshot 3: command boundary

```python
# src/constellation/__main__.py
from .reports import observation_line


def main():
    """Display one example constellation observation."""
    print(observation_line("lyra", -15))


if __name__ == "__main__":
    main()
```

For every snapshot:

1. run `git status --short`;
2. inspect the relevant files and `git diff`;
3. stage named paths;
4. inspect `git diff --staged`;
5. commit with a message describing the one change; and
6. inspect `git log --oneline --decorate`.

Then create `report-header`, add a header in the report output, and keep the branch unmerged. Explain which commit each branch points to.

Compare the evidence expected from one clean history

```text
* <newest> (HEAD -> report-header) Add night report header
* <third>  (main) Add package execution boundary
* <second> Add normalized observation report
* <first>  Create constellation package skeleton
```

Exact hashes differ because each commit identifier depends on its content and metadata. The important evidence is the parent sequence, the two branch labels, and clean `git status --short` output after each committed snapshot. `dist/`, `.venv/`, caches, and `*.egg-info/` should not enter the staged diff.

Checkpoint: inspect before recording

Know exactly what the next commit records

Choose the command or concept that matches each Git state.

1. Which command shows the content currently proposed for the next commit?

- **A.** `git log --all`
- **B.** `git diff --staged`
- **C.** `git init`
- **D.** `git branch --show-current`

2. You stage `README.md` and then edit it again. What is true?

- **A.** The later edit is automatically staged.
- **B.** The commit has already happened.
- **C.** The index can contain the earlier content while the working tree contains the later edit.
- **D.** Git deletes the earlier staged content.

3. At this level, what is a Git branch best understood as?

- **A.** A remote backup
- **B.** An ignored directory
- **C.** A complete second computer
- **D.** A movable name pointing to a commit

Write your choices before checking the answer key at the end of this book.

Audit the project as a new collaborator

Close files and begin at the repository root. Without relying on editor tabs, answer these questions from the tree and Git:

```bash
git status --short --ignored
git log --oneline --decorate --graph --all -n 8
git ls-files
```

- Can you identify importable product source without opening checks or build output?
- Can you name the supported import package without including `src`?
- Does `git ls-files` contain only deliberate source, configuration, documentation, and checks?
- Are environment, cache, installed-metadata, and build paths either absent or reported as ignored?
- Does each commit subject match its staged patch?
- Which commit do `main` and `report-header` currently name?

Use Python to classify tracked paths by their first component:

```python
import subprocess
from collections import Counter
from pathlib import Path

result = subprocess.run(
    ["git", "ls-files"],
    text=True,
    capture_output=True,
```

PYTHON (CONTINUED)

```
    check=True,
)
tracked = [Path(line) for line in result.stdout.splitlines()]
groups = Counter(path.parts[0] for path in tracked if path.parts)
print(groups)
```

The counts are descriptive rather than pass/fail rules. An unexpected `.venv`, `dist`, or cache group deserves investigation; an unfamiliar source folder may be a valid project decision. Explain every top-level group in one sentence.

Finally, clone-like confidence requires more than a clean status. In Lesson 6, a new environment will install declared source and run it away from the repository. This audit establishes that the repository contains the inputs for that proof rather than machine history masquerading as source.

Do the audit again from a plain terminal, not only an editor's source-control panel. Graphical tools are useful, but the command output creates portable evidence another learner can compare. Record the project root and branch name with that output so a clean status from the wrong repository cannot be mistaken for the intended proof. Keep that compact audit with the handoff notes for later comparison.

8. Key points for navigable, versioned projects

- A useful tree separates importable source, checks, human documentation, project metadata, and generated/local artifacts.
- In a src layout, `src` is not part of the import name. Installation makes the package importable and exposes accidental repository-root imports.
- Pass paths or configuration through explicit interfaces; do not infer a repository by walking a fixed number of parents from `__file__`.
- `.gitignore` keeps reproducible local artifacts out of untracked-path reports; it does not automatically untrack existing content.
- Git distinguishes working-tree content, staged content, and committed snapshots. Inspect each state before moving to the next.
- `git add` stages current content, so edit-after-stage creates both staged and unstaged changes.
- A commit is a local snapshot with a parent relationship. A branch is a movable name pointing into that history.

References

- PyPA discussion of src layout and flat layout
- Python package resource access with `importlib.resources`
- Pro Git: what is Git?
- Pro Git: recording changes
- Software Carpentry: tracking changes with Git
- `gitignore` pattern documentation

Give Each Project a Clean Python Environment

Online lesson and runnable notebook

Create, identify, use, inspect, and recreate isolated Python environments while distinguishing direct, transitive, runtime, and development dependencies.

- **Level:** Python Foundations
- **Estimated time:** 4–5 hours
- **You will learn:** Create a project environment, prove interpreter and installer identity, install through that interpreter, classify dependency relationships, compare declarations with environment snapshots, and recreate rather than move an environment.
- **Practice in:** A local terminal and disposable project; selected identity checks also run in notebooks

Project A needs one set of installed distributions; Project B needs another. Installing everything into one shared Python makes both projects depend on invisible machine history. A virtual environment gives one project an isolated Python installation area. It does not replace project metadata, prove that the right interpreter ran, or make an environment portable by magic.

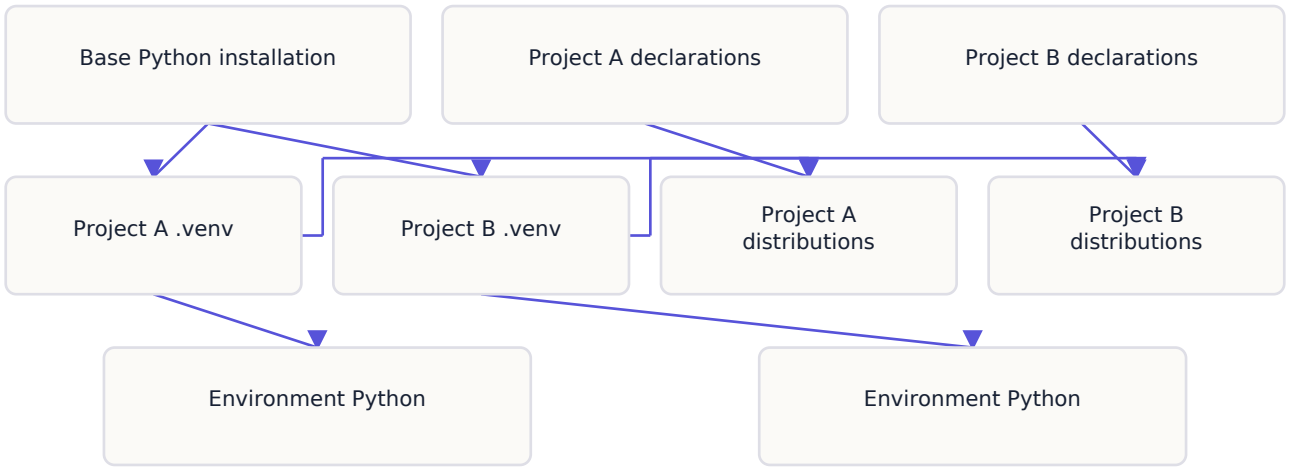
You will create two disposable environments for the constellation project and answer:

- Which interpreter created and runs the environment?
- What changes when an environment is activated?
- Why is `environment_python -m pip` stronger than trusting a prompt label?
- Which dependencies should the project declare directly?
- How do declarations, constraints, environment snapshots, and lock files differ?
- Why should a broken or moved environment normally be recreated?

1. An environment connects one project to one interpreter context

A virtual environment contains or references a Python executable and has its own installation directories. Packages installed there do not enter another ordinary virtual environment.

Two projects can begin from the same base Python while receiving separate interpreters, installers, and installed distributions.



Isolation prevents ordinary cross-environment package visibility. It does not isolate files, network access, operating-system libraries, or every environment variable.

Before creating anything, identify the current interpreter:

```python
import sys

print("executable:", sys.executable)
print("version:", sys.version.split()[0])
print("prefix:", sys.prefix)
print("base prefix:", sys.base_prefix)
```

Outside a typical virtual environment, `sys.prefix` and `sys.base_prefix` are usually equal. Inside one, `sys.prefix` points at the environment while `sys.base_prefix` points at the base installation used to create it.

This check is more dependable than reading the terminal prompt:

```python
import sys

inside_virtual_environment = sys.prefix != sys.base_prefix
print("inside virtual environment:", inside_virtual_environment)
```

Some environment tools use different mechanisms, so this exact comparison is a `venv`-focused diagnostic rather than a universal environment detector.

2. Create `.venv` with the intended Python

From the `constellation-report` project root, run:

```bash
python -m venv .venv
```

This says: ask the `python` resolved by the shell to run the standard-library `venv` module and create an environment at `.venv`. Check `python --version` first if the project requires a particular Python line.

`.venv` is a common local name because editors recognize it and one ignore rule is simple. The name is convention, not Python syntax. Keep it in `.gitignore`:

```gitignore
.venv/
```

Virtual environments contain machine-specific paths and installed artifacts. Do not commit them. Commit the project declarations used to create them.

Locate the environment interpreter portably

The executable path differs by platform:

```python
from pathlib import Path
import os


def environment_python(environment_directory):
    """Return the conventional Python executable inside a venv directory."""
    root = Path(environment_directory)
    if os.name == "nt":
        return root / "Scripts" / "python.exe"
    return root / "bin" / "python"


print(environment_python(".venv"))
```

On POSIX systems the result is `.venv/bin/python`; on Windows it is `.venv\Scripts\python.exe`. Use the path shown for your platform in later commands.

Verify that executable directly:

```bash
.venv/bin/python -c "import sys; print(sys.executable); print(sys.prefix); print(sys.base_prefix)"
```

Windows PowerShell equivalent:

```powershell
.venv\Scripts\python.exe -c "import sys; print(sys.executable); print(sys.prefix); print(sys.base_prefi ↪
    x)"
```

The output path should resolve inside `.venv`, and prefix should differ from base prefix.

Create and inspect an environment from Python

This complete example uses a temporary workspace and never depends on shell activation:

```python
import os
import subprocess
import sys
from pathlib import Path
from tempfile import TemporaryDirectory


def environment_python(environment_directory):
    root = Path(environment_directory)
    relative = Path("Scripts/python.exe") if os.name == "nt" else Path("bin/python")
    return root / relative


with TemporaryDirectory() as temporary_name:
    environment = Path(temporary_name) / "practice-env"
```

PYTHON (CONTINUED)

```
    subprocess.run([sys.executable, "-m", "venv", str(environment)], check=True)
    python = environment_python(environment)
    result = subprocess.run(
        [
            str(python),
            "-c",
            "import sys; print(sys.executable); "
            "print(sys.prefix != sys.base_prefix)",
        ],
        text=True,
        capture_output=True,
        check=True,
    )
    print(result.stdout)
```

The child process identifies itself. No prompt decoration is involved.

Checkpoint: prove environment identity

Identify the Python that will actually run

Choose the observation that proves interpreter identity.

1. Which check most directly proves that a command used `.venv`?

- **A.** Run `.venv`'s Python and print `sys.executable` and prefixes.
- **B.** Look only for `(.venv)` in the prompt.
- **C.** Check that a `.venv` directory name exists.
- **D.** Run an unrelated system command.

2. What does an ordinary virtual environment primarily isolate?

- **A.** All files and network access
- **B.** Python interpreter context and installed distributions
- **C.** Git history
- **D.** Operating-system accounts

3. What should normally be committed instead of the `.venv` directory?

- **A.** The interpreter binary
- **B.** Every cache file
- **C.** Project metadata and dependency declarations
- **D.** The terminal prompt

Write your choices before checking the answer key at the end of this book.

3. Activation changes command lookup; it does not create the environment

Activation prepends the environment's executable directory to the current shell's command search path and usually changes the prompt. Common commands are:

```bash
# macOS, Linux, and many POSIX shells
source .venv/bin/activate
```

```powershell
# Windows PowerShell
.venv\Scripts\Activate.ps1
```

After activation, `python` should resolve to the environment executable:

```bash
python -c "import sys; print(sys.executable)"
python -m pip --version
```

Activation is convenient for repeated interactive commands. It is not required. Scripts, editors, CI jobs, and careful diagnostics can invoke the environment's Python by full path.

The prompt can mislead when:

- a shell startup script customizes it;
- an editor launches a different interpreter;
- another activation changes lookup order;
- a copied prompt is shown in instructions; or
- a command is run in a different terminal.

Trust `sys.executable` and `python -m pip --version`, which reports both pip's version and installation location.

Deactivate only changes the current shell's lookup configuration:

```bash
deactivate
```

It does not delete the environment or uninstall its packages.

4. Install through the interpreter that should receive the package

Prefer this shape:

```bash
python -m pip install SOME_DISTRIBUTION
```

When activated, first verify `python`. Without activation, use the explicit environment path:

```bash
.venv/bin/python -m pip install SOME_DISTRIBUTION
```

The selected interpreter loads its `pip` module. A bare `pip` executable can be associated with another Python on the shell path.

Before adding anything, inspect the environment without network access:

```bash
.venv/bin/python -m pip --version
.venv/bin/python -m pip list
.venv/bin/python -m pip check
```

- `pip --version` exposes pip's path and Python association.
- `pip list` reports installed distributions.
- `pip check` reports incompatible or missing requirements among installed distributions; silence means it found no such inconsistency, not that the whole application works.

Observe isolation with metadata

```python
from importlib.metadata import distributions

installed = sorted(
    distribution.metadata["Name"]
    for distribution in distributions()
    if distribution.metadata["Name"]
)
print(installed[:10])
```

Run it with base Python and environment Python. The lists may differ. A newly created `venv` normally includes pip unless created with `--without-pip`, but the exact bootstrap versions depend on the Python installation.

Do not teach environment isolation by installing random packages globally. Use disposable environments and delete them when finished.

Run an interpreter mismatch clinic

When installation succeeds but import fails, compare the installer report and runtime in the same terminal:

```bash
python -c "import sys; print(sys.executable)"
python -m pip --version
python -m pip show pip
```

All reported locations should belong to the intended context. Classify the evidence before repairing:

Evidence	Likely explanation	Next controlled check
`sys.executable` is outside `.venv`	shell/editor selected another Python	invoke `.venv` Python by full path
pip path is outside the printed Python's environment	command spelling mixed launchers	use that Python with `-m pip`
distribution appears in `pip show`, but import name fails	distribution/import names may differ	read primary package usage docs and installed metadata
module origin is a project file with the same name	local shadowing	print `module.__file__` from a fresh process
package imports in one kernel after uninstall	cached module remains in memory	restart and rerun the exact import

Do not respond to all five cases with another `pip install`. Repeated installation into the wrong interpreter can report success forever while the runtime remains unchanged.

Capture both standard channels when a child interpreter fails:

```python
import subprocess
import sys

result = subprocess.run(
    [sys.executable, "-m", "pip", "show", "definitely-not-installed"],
    text=True,
    capture_output=True,
)
print("return code:", result.returncode)
print("stdout:", result.stdout)
print("stderr:", result.stderr)
```

The exact wording can vary across pip releases, so build the diagnosis around the return code, selected interpreter, and distribution identity rather than a fragile substring alone.

Checkpoint: route installations deliberately

Send commands to the intended environment

Choose the command or interpretation with the strongest identity link.

1. Why prefer `.venv/bin/python -m pip ...` during diagnosis?

- **A.** It installs without an environment.
- **B.** It guarantees every release is compatible.
- **C.** It skips distribution metadata.
- **D.** It explicitly connects pip to the environment interpreter being invoked.

2. What is activation mainly doing for the current shell?

- **A.** Changing command lookup so the environment executables are found first
- **B.** Building a wheel
- **C.** Committing installed packages
- **D.** Moving the environment to another computer

3. `python -m pip check` reports no problems. What has been established?

- **A.** Every application behavior is correct.
- **B.** Installed requirements have no inconsistency that pip detected.
- **C.** The wheel contains all project files.
- **D.** The environment is portable.

Write your choices before checking the answer key at the end of this book.

5. Declare direct needs, not the entire environment accident

Dependency words describe different relationships:

Kind	Meaning	Constellation example
Direct	The project intentionally imports or invokes it	a future astronomy library imported by project source
Transitive	A direct dependency needs it	a parsing library needed internally by that astronomy distribution
Runtime	Users need it to run product behavior	the astronomy library if report calculation imports it
Development	Maintainers need it to build, check, or document	`build`, pytest, Ruff, or a documentation generator
Optional	Needed only for a named extra feature	plotting support not used by the base report

Declare direct project requirements. Do not copy every transitive package from `pip freeze` into runtime dependencies. The direct distribution owns its own requirements, and duplicating the entire graph hides which relationships your project actually chose.

For the current constellation package, all runtime behavior uses Python's standard library. The runtime dependency list can be empty. The repository will use `build` during development in Lesson 6, so `build` is a development tool—not runtime baggage for users of `constellation`.

Derive direct dependencies from source responsibilities

Start with project-owned import statements and external processes. Classify each top-level name:

```python
# Standard-library imports: no distribution declaration is added for these.
from collections import Counter
from pathlib import Path

# Project-package import: delivered by this distribution itself.
# from constellation import observation_line

# Hypothetical third-party import: map it to its providing distribution.
# import astronomy_catalog
```

Do not turn the text after every `import` into a pip name mechanically. A name may be standard library, project source, or supplied by a differently named distribution. Confirm the third-party mapping in that project's primary installation documentation and installed metadata.

Then ask where the import occurs. If product source imports `astronomy_catalog`, users need its distribution at runtime. If only a local build script imports `build`, it is development tooling. If a tutorial notebook uses plotting but the package does not, decide whether that notebook is a supported development environment, an optional extra, or merely an external example. Location alone is a clue; the documented product workflow supplies the final classification.

Transitive packages can appear in `pip list` even though no project source names them. Removing one manually may break the direct dependency that selected it. Let the installer resolve declared relationships, and use `python -m pip check` after controlled changes.

Read a dependency specification

Examples of distribution requirement strings are:

```text
example-reader>=2
example-reader>=2,<3
example-reader~=2.4
example-reader; python_version < "3.12"
```

These are demonstrations of syntax, not recommendations for a real distribution named `example-reader`.

- `>=2` sets a lower bound but allows future major releases.
- `>=2,<3` sets an inclusive lower and exclusive upper range.
- `~=2.4` expresses compatible release behavior as defined by the packaging specification.
- the marker applies the requirement only when its condition is true.

Choose bounds from tested compatibility and project policy, not habit. == can be useful in a controlled application lock but is often too restrictive for a reusable library dependency declaration. Leaving every dependency unbounded is also a policy choice with risks.

6. Declarations, constraints, snapshots, and locks answer different questions

These artifacts are related but not interchangeable:

Project declaration

`pyproject.toml` says what direct distributions and Python versions the project requires. It is maintained as part of source and used when building/installing.

Environment snapshot

```bash
python -m pip freeze
```

This reports installed distribution versions in the current environment. It is useful diagnostic evidence, but it may include unrelated, transitive, editable, or manually installed content. It does not explain why each item belongs.

Constraints file

A pip constraints file limits versions considered during an installation but does not itself request installation of every listed project. It is an environment-control input, not a package’s runtime metadata.

Tool-specific lock file

Some project managers resolve dependencies and record a repeatable set in a lock format. Lock semantics, supported targets, and installation commands belong to that tool. Python packaging does not currently make every lock file universal across every workflow.

Use precise statements: “this is the project’s declared range,” “this is what was installed here,” or “this tool resolved this lock.” Avoid saying any one of them is simply “the dependencies” without context.

Compare declarations with observations

`importlib.metadata.requires` reads an installed distribution's recorded requirements:

```python
from importlib.metadata import PackageNotFoundError, requires

try:
    requirements = requires("pip") or []
    print(requirements[:5])
except PackageNotFoundError:
    print("pip distribution metadata was not found")
```

This reports metadata from this installed environment. It does not edit `pyproject.toml` and does not decide what your project should declare.

7. Recreate environments instead of moving or repairing them forever

Virtual environments embed absolute paths in scripts and are generally not portable. If the project directory moves, Python is upgraded, or the environment becomes mysterious, recreation is usually safer:

```bash
# Run only after leaving/deactivating the environment.
rm -rf .venv
python -m venv .venv
.venv/bin/python -m pip install --upgrade pip
```

On Windows, remove `.venv` through PowerShell or the file explorer, then use the Windows interpreter path. In an offline or tightly controlled environment, do not upgrade from the network casually; use the organization's approved package source and versions.

After Lesson 6 supplies project metadata, the recreation flow continues:

```bash
.venv/bin/python -m pip install --editable .
.venv/bin/python -m pip check
.venv/bin/python -c "import constellation; print(constellation.__file__)"
```

The declarations and source are durable. `.venv` is disposable.

Run a two-environment identity lab

The following creates two environments without third-party downloads:

```python
import os
import subprocess
import sys
from pathlib import Path
from tempfile import TemporaryDirectory


def environment_python(environment_directory):
    root = Path(environment_directory)
    relative = Path("Scripts/python.exe") if os.name == "nt" else Path("bin/python")
    return root / relative
```

PYTHON (CONTINUED)

```
with TemporaryDirectory() as temporary_name:
    root = Path(temporary_name)
    environments = [root / "alpha", root / "beta"]

    for environment in environments:
        subprocess.run([sys.executable, "-m", "venv", str(environment)], check=True)

    identities = []
    for environment in environments:
        python = environment_python(environment)
        result = subprocess.run(
            [str(python), "-c", "import sys; print(sys.executable)"],
            text=True,
            capture_output=True,
            check=True,
        )
        identities.append(Path(result.stdout.strip()).resolve())

    assert identities[0] != identities[1]
    assert all(identity.exists() for identity in identities)
    print(*identities, sep="\n")
```

Extend the lab:

1. run `-m pip --version` through each interpreter;
2. print `sys.prefix` and `sys.base_prefix` in each;
3. record the output in a table;
4. delete `alpha` after its process exits;
5. recreate `alpha`; and
6. explain why its path can match the old spelling even though it is a new environment.

Compare a compact environment evidence helper

PYTHON

```
import json
import subprocess


def environment_evidence(python_executable):
    """Return identity evidence reported by one Python interpreter."""
    program = (
        "import json, sys; "
        "print(json.dumps({"
        "'executable': sys.executable, "
        "'prefix': sys.prefix, "
        "'base_prefix': sys.base_prefix, "
        "'version': sys.version.split()[0]}))"
    )
    result = subprocess.run(
        [str(python_executable), "-c", program],
        text=True,
        capture_output=True,
        check=True,
    )
    return json.loads(result.stdout)
```

The function trusts the child interpreter’s report, not the parent process’s assumption. A recreated directory can reuse a path, but its installed state is new and must be restored from declarations.

Checkpoint: explain and recreate the dependency context

Keep declarations and observations distinct

Choose the artifact or dependency relationship described.

1. Project source imports a third-party astronomy package, which itself installs a parser. What should the project normally declare directly?

- **A.** Every distribution ever installed
- **B.** Only the parser
- **C.** The astronomy distribution it intentionally uses
- **D.** The `.venv` directory

2. What does `python -m pip freeze` primarily report?

- **A.** Why every dependency was chosen
- **B.** The only valid project metadata
- **C.** A universal cross-tool lock
- **D.** Installed distribution versions observed in the current environment

3. A project and its `.venv` were moved, and launchers contain old paths. What is the normal repair?

- **A.** Recreate the environment from the intended Python and project declarations.
- **B.** Commit and share the broken environment.
- **C.** Edit every generated script manually.
- **D.** Add random directories to `sys.path`.

Write your choices before checking the answer key at the end of this book.

Write a recreation note another learner can follow

Finish with a short environment handoff that contains commands and expected identity evidence, not a copy of your prompt:

```text
Supported Python: 3.10 or newer for this exercise
Environment path: .venv (ignored and disposable)
Create:           python -m venv .venv
Install project:  .venv/bin/python -m pip install --editable .
Verify identity:  .venv/bin/python -c "import sys; print(sys.executable)"
Check metadata:   .venv/bin/python -m pip check
Run:              .venv/bin/python -m constellation
```

Provide the Windows executable spelling beside this POSIX version when the audience uses both platforms. Do not paste `pip freeze` as though it explains the project. If a tool-specific lock is part of the chosen workflow, name the tool and the exact lock-based install command.

Test the note yourself from an empty environment directory. A useful handoff lets a learner answer:

- which base Python should create the environment;
- which path is safe to delete and recreate;

- which command installs declared project needs;
- which command proves the runtime interpreter;
- which command checks installed requirement consistency; and
- which behavior proves the project is ready for the next packaging step.

Record any package-index or offline-cache assumption. An environment that recreates only while connected to your private machine is not yet a complete handoff. Do not include secrets, local access tokens, or environment binaries in the note or repository.

Delete the practice environment, follow only the note, and compare the new interpreter and distribution list with the old evidence. Paths and bootstrap tool versions may differ while the declared project behavior remains the same. That is the intended distinction between reproducible behavior and identical machine state.

8. Key points for clean environments

- A virtual environment isolates a Python interpreter context and installation directories; it is not a container or a complete security boundary.
- The creating Python matters. Verify the environment through its own `sys.executable`, `sys.prefix`, and `sys.base_prefix`.
- Activation changes shell command lookup for convenience. It is not required and its prompt is not proof.
- Invoke pip through the interpreter that should receive the installation.
- Classify direct, transitive, runtime, development, and optional relationships; do not declare an accidental snapshot as runtime requirements.
- Project declarations, constraints, environment snapshots, and tool-specific locks answer different questions.
- Do not commit or move `.venv`. Recreate it from the intended Python and deliberate project inputs.

References

- Python `venv` documentation
- PyPA tutorial: installing packages in a virtual environment
- pip user guide: running pip
- PyPA dependency-specifier specification
- pip requirements and constraints files
- `importlib.metadata` distribution metadata

Build and Verify an Installable Python Project

Online lesson and runnable notebook

Declare standards-based project metadata, install editable source, create and inspect an sdist and wheel, and prove the wheel works in a clean environment outside the repository.

- **Level:** Python Foundations
- **Estimated time:** 5–6 hours
- **You will learn:** Read the main `pyproject.toml` tables, connect source and distribution identities, install editable source, expose module and console entry points, build and inspect artifacts, and verify the exact wheel in a fresh environment.
- **Practice in:** A local editor and terminal with two disposable virtual environments

The constellation source works in its project environment. That is not yet a release proof. Another person receives an installable **distribution**, not your working tree, editor settings, or accidental current-directory imports. This lesson writes one complete project description, builds the two common artifacts, and tests the wheel from somewhere that cannot see `src/`.

Keep these questions beside the terminal:

- Which metadata is standardized, and which table belongs to a particular tool?
- Which name is used by installers, imports, and console commands?
- What changes between editable and regular installation?
- What does a build frontend ask a build backend to do?
- Which files actually entered the wheel?
- Can the exact wheel run in a new environment outside the repository?

1. One `pyproject.toml` describes the installable project

At the root of `constellation-report`, replace the placeholder with:

TOML

```toml
[build-system]
requires = ["hatchling>=1.26"]
build-backend = "hatchling.build"

[project]
name = "constellation-report"
version = "0.1.0"
description = "Build small constellation observation reports."
readme = "README.md"
requires-python = ">=3.10"
authors = [
  { name = "Course Learner" }
]
dependencies = []

[project.scripts]
constellation-report = "constellation.cli:main"
```

TOML (CONTINUED)

```
[tool.hatch.build.targets.wheel]
packages = ["src/constellation"]
```

TOML uses tables, keys, strings, arrays, and inline tables. The spelling and value type matter. This file has four responsibilities:

Table	Who defines it?	What it says here
`[build-system]`	Python packaging standard	backend interface and requirements needed to build
`[project]`	Python packaging standard	distribution identity, version, compatibility, description, and dependencies
`[project.scripts]`	Python packaging standard	installed command name mapped to an importable callable
`[tool.hatch.build.targets.wheel]`	Hatchling	which src-layout package the chosen backend should put in the wheel

The last table is tool-specific. Replacing Hatchling with another backend would require reading that backend's configuration; do not assume all `[tool.*]` tables are portable standards.

Read each identity precisely

TEXT

```
Repository directory: constellation-report/
Distribution name:    constellation-report
Import package:       constellation
Module:               constellation.reports
Console command:      constellation-report
```

The repository directory can be renamed without changing installed metadata. The distribution name is used by installers and `importlib.metadata`. The import-package name is used by Python. The console command is a generated launcher whose name happens to match the distribution here, but it need not.

State Python compatibility honestly

`requires-python = ">=3.10"` tells installers which Python versions the release claims to support. It does not install Python and does not test those versions. Choose the range from syntax used, dependencies, and actual verification. A mistyped range can make an otherwise correct wheel impossible to install.

Runtime dependencies stay empty when runtime uses only Python

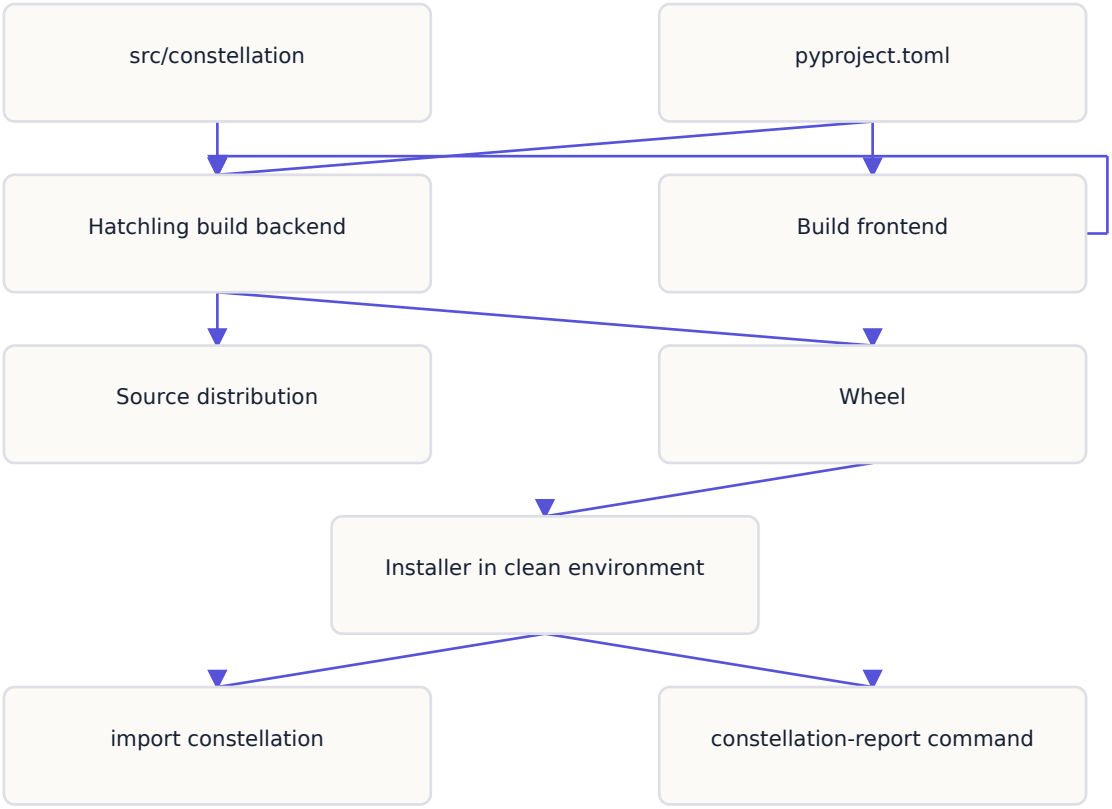
The constellation package imports only standard-library modules, so:

TOML

```
dependencies = []
```

is accurate. Hatchling and `build` are development/build tools. Users do not need to import them when running the installed report, so they do not belong in `[project].dependencies`.

Standard metadata connects the source tree to build tooling and the artifact a user installs.



Build frontend → Hatchling build backend: standard build request

Checkpoint: read the project tables

Connect metadata to its packaging role

Choose the table or identity described.

1. What does `[build-system]` primarily declare?

- **A.** The report's runtime observations
- **B.** The backend interface and requirements needed to build
- **C.** Every package installed on the machine
- **D.** Git author identity

2. The installed package uses only the standard library at runtime. Where should `build` be listed?

- **A.** As an import package alias
- **B.** In every wheel filename
- **C.** As development tooling, not `[project].dependencies`
- **D.** As the project's console command

3. Which statement correctly distinguishes `constellation-report` and `constellation`?

- **A.** They must be the same valid Python identifier.

- **B.** The first is a Git branch and the second is a filename.
- **C.** The first is a Python version specifier.
- **D.** The first is the distribution/command spelling here; the second is the import package.

Write your choices before checking the answer key at the end of this book.

2. Give every execution route one callable

Add a small command boundary. Unit 12 will teach full argument parsing, streams, and exit codes; this lesson only connects packaging entry points safely.

PYTHON
```
# src/constellation/cli.py
from .reports import observation_line


def main():
    """Display one example observation report."""
    print(observation_line("moon gate", -15))
```

Make package-module execution delegate to the same callable:

PYTHON
```
# src/constellation/__main__.py
from .cli import main


if __name__ == "__main__":
    main()
```

There are now three intended entry routes:

TEXT
```
Python API:       from constellation import observation_line
Module command:   python -m constellation
Installed command: constellation-report
```

The first calls functions explicitly. The second executes `__main__.py`. The third is generated from this metadata:

TOML
```
[project.scripts]
constellation-report = "constellation.cli:main"
```

The right side means: import the module `constellation.cli`, retrieve its `main` attribute, and call it. Do not add parentheses. This is metadata, not a Python call expression.

Keep `cli.py` import-safe

Do not call `main()` unconditionally in `cli.py`:

PYTHON
```
# Wrong design: importing constellation.cli would run the command.
def main():
    print("Moon Gate: 345°")
```

```
PYTHON (CONTINUED)
# main()  # Do not place this unguarded call in the real file.
```

The generated launcher calls `main`. `__main__.py` calls `main` only when module mode executes it. An ordinary import merely defines it.

3. Install editable source for development

Activate `.venv` and verify its identity, or use its Python explicitly. From the project root:

```
BASH
.venv/bin/python -m pip install --editable .
```

PowerShell uses `.venv\Scripts\python.exe`. The dot means “install the project described by `pyproject.toml` in the current directory.”

An **editable installation** records a development link so imports use the working source. Editing `src/constellation/reports.py` can affect the next fresh Python process without rebuilding and reinstalling a wheel. Backend details vary, so treat editable installation as a development mode—not as the release artifact users receive.

Prove origin, metadata, API, and both entry routes:

```
BASH
.venv/bin/python -c "import constellation; print(constellation.__file__)"
.venv/bin/python -c "from importlib.metadata import version; print(version('constellation-report'))"
.venv/bin/python -c "from constellation import normalize_degrees; print(normalize_degrees(-1))"
.venv/bin/python -m constellation
.venv/bin/constellation-report
```

On Windows, the generated console launcher is normally `.venv\Scripts\constellation-report.exe`.

Expected important values are:

```
TEXT
.../src/constellation/__init__.py
0.1.0
359
Moon Gate: 345°
Moon Gate: 345°
```

The editable origin points into `src`. This proves the development installation is connected, not that a wheel contains the package.

Diagnose an editable install failure in order

1. Confirm the command ran from the directory containing `pyproject.toml`.
2. Confirm `sys.executable` belongs to `.venv`.
3. Read the first build/installation error, not only the final summary.
4. Validate table spelling and value types in `pyproject.toml`.
5. Confirm `src/constellation/__init__.py` and the backend package path agree.
6. After success, start a fresh Python process and print module origin.

Do not “repair” a metadata error with `sys.path.insert`. That bypasses the contract under investigation.

Checkpoint: connect editable source and entry points

Verify development installation without confusing it with release

Choose what each command or metadata entry proves.

1. After an editable installation, `constellation.__file__` **points into the repository's** `src` **directory. What does this show?**

- **A.** The development installation resolves imports to working source.
- **B.** The wheel was clean-installed elsewhere.
- **C.** Git committed the package.
- **D.** Every dependency is locked.

2. What does `constellation.cli:main` **in** `[project.scripts]` **identify?**

- **A.** A filesystem path to execute directly
- **B.** An importable module and callable attribute
- **C.** A distribution version range
- **D.** A Git remote

3. Why should `__main__.py` **and the console script delegate to the same** `main` **callable?**

- **A.** Python permits only one source file.
- **B.** It disables package imports.
- **C.** It avoids duplicated command behavior drifting between entry routes.
- **D.** It turns the wheel into a source distribution.

Write your choices before checking the answer key at the end of this book.

4. Build frontends and backends have different jobs

Python packaging tools cooperate through standardized interfaces:

- An **installer** such as pip installs distributions and resolves requirements.
- An **environment creator/manager** creates or selects interpreter contexts.
- A **build frontend** such as `build` asks for artifacts through standard hooks.
- A **build backend** such as Hatchling implements those hooks for a project.

One tool can cover several roles, but the roles remain useful for reasoning. In this workflow:

```text
python -m venv .venv          environment creation
python -m pip install ...     installation
python -m build               build frontend
hatchling.build               configured backend
```

Install the frontend as development tooling in the project environment:

```bash
.venv/bin/python -m pip install "build>=1"
```

Then remove old artifacts and build from the repository root:

```bash
rm -rf dist build
.venv/bin/python -m build
```

By default, `build` creates an isolated build environment, installs the `[build-system].requires` there, asks the backend for a source distribution, and builds a wheel (normally from that source distribution). This can require access to a configured package index unless requirements are already available.

Do not put `build` into runtime dependencies merely because maintainers use it. The finished wheel records only `[project].dependencies` as runtime requirements.

5. An sdist and a wheel serve different installation paths

After a successful build, `dist/` should contain files resembling:

```text
constellation_report-0.1.0-py3-none-any.whl
constellation_report-0.1.0.tar.gz
```

Filename normalization can change hyphens to underscores. Read metadata rather than inferring every identity rule from the filename.

A **source distribution** (`.tar.gz`) contains source material and metadata from which a backend can build. A **wheel** (`.whl`) is a built distribution with a standard archive layout that installers can place without rebuilding project source in the usual pure-Python case.

Neither artifact is “the Git repository zipped up.” Backend selection rules decide included files. Inspect them.

Inspect the source distribution without extracting it

Use `tarfile` to list the `.tar.gz` safely. Listing avoids overwriting files in the current directory:

```python
from pathlib import Path
from tarfile import open as open_tar

source_archives = sorted(Path("dist").glob("*.tar.gz"))
if len(source_archives) != 1:
    raise RuntimeError(
        f"expected one source distribution, found {len(source_archives)}"
    )

source_archive = source_archives[0]
with open_tar(source_archive, mode="r:gz") as archive:
    source_names = sorted(member.name for member in archive.getmembers())

for name in source_names:
    print(name)
```

The archive should contain project metadata, README material, and package source needed for another build. It need not include `.git`, `.venv`, caches, or a previously generated `dist/` directory. Treat any archive from an untrusted source as external input; listing is safer than extracting, and safe extraction still requires attention to member paths and links.

Why inspect both artifacts? The default frontend workflow can build the wheel from the sdist. A source file present in the repository but absent from the sdist may therefore disappear from the wheel even if a direct local wheel build would have found it. Checking both makes the handoff chain visible.

Inspect wheel contents with the standard library

PYTHON

```
from pathlib import Path
from zipfile import ZipFile

wheels = sorted(Path("dist").glob("*.whl"))
if len(wheels) != 1:
    raise RuntimeError(f"expected one wheel, found {len(wheels)}")

wheel = wheels[0]
with ZipFile(wheel) as archive:
    names = sorted(archive.namelist())

for name in names:
    print(name)
```

Look for:

TEXT

```
constellation/__init__.py
constellation/__main__.py
constellation/angles.py
constellation/cli.py
constellation/reports.py
...dist-info/METADATA
...dist-info/WHEEL
...dist-info/entry_points.txt
```

Do not expect `.venv/`, `.git/`, `checks/`, caches, or `dist/` inside the product wheel. The README may be represented in distribution metadata rather than as a top-level package file.

Read metadata from inside the wheel

PYTHON

```
from email.parser import BytesParser
from email.policy import default
from pathlib import Path
from zipfile import ZipFile

wheel = next(Path("dist").glob("*.whl"))
with ZipFile(wheel) as archive:
    metadata_name = next(
        name for name in archive.namelist() if name.endswith(".dist-info/METADATA")
    )
    metadata = BytesParser(policy=default).parsebytes(archive.read(metadata_name))

print(metadata["Name"])
print(metadata["Version"])
print(metadata["Requires-Python"])
print(metadata.get_all("Requires-Dist", []))
```

Expected values are `constellation-report`, `0.1.0`, `>=3.10`, and an empty runtime requirement list. This is artifact evidence, not merely the source TOML you hoped the backend would use.

Reject ambiguous artifact selection

Avoid this shortcut:

```python
from pathlib import Path

# Avoid: `next` silently chooses whichever matching artifact appears first.
possibly_stale_wheel = next(Path("dist").glob("*.whl"))
```

If `0.1.0` and `0.2.0` wheels both remain, a successful install may verify the wrong release. Clean the output directory before building and assert the exact number of artifacts afterward:

```python
from pathlib import Path

wheels = sorted(Path("dist").glob("*.whl"))
sdists = sorted(Path("dist").glob("*.tar.gz"))

assert len(wheels) == 1, wheels
assert len(sdists) == 1, sdists
print(wheels[0].name)
print(sdists[0].name)
```

For automation beyond this lesson, pass the exact expected artifact path from the build step rather than rediscovering it through a broad glob.

6. A clean installation must not see the repository source

Create a second environment in a temporary directory, install the **exact wheel file**, and run from another directory. This helper works on POSIX and Windows:

```python
import os
import subprocess
import sys
from pathlib import Path
from tempfile import TemporaryDirectory


def environment_python(environment_directory):
    root = Path(environment_directory)
    relative = Path("Scripts/python.exe") if os.name == "nt" else Path("bin/python")
    return root / relative


project_root = Path.cwd()
wheels = sorted((project_root / "dist").glob("*.whl"))
if len(wheels) != 1:
    raise RuntimeError(f"expected one wheel, found {len(wheels)}")
wheel = wheels[0].resolve()

with TemporaryDirectory() as temporary_name:
    root = Path(temporary_name)
    clean_environment = root / "clean-env"
    outside_repository = root / "run-here"
    outside_repository.mkdir()

    subprocess.run(
        [sys.executable, "-m", "venv", str(clean_environment)],
        check=True,
    )
    clean_python = environment_python(clean_environment)
    subprocess.run(
        [str(clean_python), "-m", "pip", "install", str(wheel)],
        check=True,
    )
```

PYTHON (CONTINUED)

```
    proof = subprocess.run(
        [
            str(clean_python),
            "-c",
            "from importlib.metadata import version; "
            "import constellation; "
            "print(version('constellation-report')); "
            "print(constellation.__file__); "
            "print(constellation.normalize_degrees(-1))",
        ],
        cwd=outside_repository,
        text=True,
        capture_output=True,
        check=True,
    )
    print(proof.stdout)

    module_run = subprocess.run(
        [str(clean_python), "-m", "constellation"],
        cwd=outside_repository,
        text=True,
        capture_output=True,
        check=True,
    )
    print(module_run.stdout)
```

The installed origin should point inside the temporary clean environment, not the repository's `src`. The version should be `0.1.0`, normalization should print `359`, and module execution should print `Moon Gate: 345°`.

To test the generated console launcher portably, locate its scripts directory:

PYTHON

```
from pathlib import Path
import os


def environment_command(environment_directory, command_name):
    root = Path(environment_directory)
    if os.name == "nt":
        return root / "Scripts" / f"{command_name}.exe"
    return root / "bin" / command_name
```

Then run `environment_command(clean_environment, "constellation-report")` through `subprocess.run`. The launcher should produce the same output as module mode.

Compare installed metadata with source and wheel metadata

Ask the clean environment—not the development process—to report what it received:

PYTHON

```
import subprocess

metadata_program = (
    "from importlib.metadata import distribution; "
    "d = distribution('constellation-report'); "
    "print(d.metadata['Name']); "
    "print(d.version); "
    "print(d.metadata['Requires-Python']); "
    "print(d.requires or [])"
)
```

PYTHON (CONTINUED)

```
result = subprocess.run(
    [str(clean_python), "-c", metadata_program],
    cwd=outside_repository,
    text=True,
    capture_output=True,
    check=True,
)
print(result.stdout)
```

The values should agree with the wheel inspection. This three-way comparison catches different mistakes:

- source TOML wrong: repair project metadata before rebuilding;
- source correct but wheel metadata stale: clean artifacts and rebuild;
- wheel correct but clean environment reports another version: verify the exact installed path and environment interpreter;
- metadata correct but import origin points to the repository: remove the source-tree shortcut and rerun outside it.

Do not import a `__version__` constant and assume it matches distribution metadata unless the project deliberately has one version source and tests the relationship. This lesson uses installed distribution metadata as the release version record.

Preserve subprocess failures as release evidence

During a clean run, do not use `check=True` until you know which output you need to inspect. Capture the first failing attempt:

PYTHON

```
import subprocess

attempt = subprocess.run(
    [str(clean_python), "-m", "constellation"],
    cwd=outside_repository,
    text=True,
    capture_output=True,
)
print("return code:", attempt.returncode)
print("stdout:", attempt.stdout)
print("stderr:", attempt.stderr)
```

A missing module, missing callable, wrong output, or duplicated output identifies a different contract. After recording it, repair source/metadata, rebuild from a clean `dist/`, recreate or reinstall into the clean environment, and rerun. An old installed wheel cannot observe a source edit by itself.

Why the outside directory matters

If the proof runs from the repository root, an accidental flat-layout package, path modification, or current-directory file might mask a broken wheel. A fresh environment plus unrelated working directory isolates the artifact contract. :::

Checkpoint: distinguish building from release proof

Inspect and verify what a user receives

Choose the strongest evidence for each artifact claim.

1. `python -m build` **succeeded. What still needs separate verification?**

- **A.** Whether Python has strings
- **B.** Whether Git exists anywhere
- **C.** Whether TOML supports tables
- **D.** Artifact contents and behavior after installing the exact wheel elsewhere

2. Which origin best supports a successful clean-wheel proof?

- **A.** `constellation.__file__` points inside the new environment, while running outside the repository.
- **B.** It points into the repository `src` directory.
- **C.** The current directory is the repository root.
- **D.** The module was already loaded in the development notebook.

3. In this workflow, how do `build` **and Hatchling relate?**

- **A.** Both are runtime report functions.
- **B.** `build` is the frontend requesting artifacts; Hatchling is the configured backend producing them.
- **C.** Hatchling creates Git commits for `build`.
- **D.** They are two import names for the constellation package.

Write your choices before checking the answer key at the end of this book.

7. Run the complete release rehearsal

Starting from a clean project checkout, produce this evidence in order:

1. **Tree:** `src/constellation`, checks, README, `.gitignore`, and `pyproject.toml` are present; generated folders are absent.
2. **Environment:** `.venv`'s Python reports its executable and differing `prefix`/`base_prefix`.
3. **Editable install:** package origin points into `src`, public assertions pass, and module/console routes agree.
4. **Build:** one sdist and one wheel exist after cleaning `dist/`.
5. **Inspection:** the wheel contains every package module and entry-point metadata but no local environment or caches.
6. **Clean install:** a second environment installs the exact wheel path without seeing repository source.
7. **Outside run:** public API, version metadata, module command, and generated console launcher all produce expected results from another directory.
8. **Git:** `git status --short` contains only deliberate source changes; build artifacts remain ignored.

Add one changed requirement: update the version to `0.2.0` and change the demo label from `moon gate` to `lunar archive`. Clean `dist/`, rebuild, and verify the new wheel. Do not leave both versions in `dist/`, because a glob that silently chooses one would make the proof ambiguous.

Compare a compact release evidence checklist

```text
[ ] Environment executable is inside .venv
[ ] Editable origin is inside repository/src
[ ] Distribution metadata version is 0.2.0
[ ] Exactly one sdist and one wheel were built
[ ] Wheel contains constellation package and entry_points.txt
[ ] Wheel contains no .venv, .git, __pycache__, checks, or dist directory
[ ] Clean origin is inside the second environment, not repository/src
[ ] API returns 359 for -1 degrees
[ ] python -m constellation prints Lunar Archive: 345°
[ ] installed constellation-report prints the same line
[ ] proof command ran outside the repository
[ ] repository status contains no generated artifacts
```

If a check fails, retain the command, return code, standard output, standard error, interpreter, working directory, and relevant origin. Repair the earliest failed contract, rebuild from clean `dist/`, and repeat the clean install.

Diagnose the release layer before editing code

Use the first failed observation to choose the next inspection:

Failure	Inspect next	Avoid
TOML parse or validation error	named table, key, line, and value type	changing package functions
backend cannot load in isolated build	`[build-system].requires`, configured index, and connectivity	adding backend to runtime dependencies
editable import missing	environment Python, install result, package path, and origin	permanent `sys.path` mutation
wheel lacks one module	backend inclusion configuration and sdist contents	copying source into the clean environment
console launcher names a missing callable	`[project.scripts]` target and import-safe module	duplicating command logic in metadata
clean install refuses Python version	`Requires-Python` in source and wheel metadata	forcing installation without understanding compatibility
outside run imports repository source	working directory, environment, editable state, and module origin	treating passing output as release proof

One repair can require rebuilding and reinstalling. Source edits do not alter an already built wheel. Wheel edits do not alter an already installed copy. Keep a simple chain in the evidence log:

```text
source commit -> pyproject metadata -> sdist -> wheel -> clean installation
```

Write the filename and SHA-256 digest beside the installed wheel when several processes could handle artifacts:

```python
from hashlib import sha256
from pathlib import Path

wheels = sorted(Path("dist").glob("*.whl"))
```

PYTHON (CONTINUED)

```
if len(wheels) != 1:
    raise RuntimeError(f"expected one wheel, found {len(wheels)}")
wheel = wheels[0]
digest = sha256(wheel.read_bytes()).hexdigest()
print(wheel.name)
print(digest)
```

The digest identifies exact bytes; it does not prove those bytes are safe or correct. Pair identity with metadata, content inspection, and behavior. Signing and full supply-chain policy belong beyond this Foundations unit.

Hand the project to a clean-room learner

Ask another learner—or simulate one with a new directory—to use only:

- the source repository at one named commit;
- a supported Python interpreter;
- documented access to build requirements;
- the release rehearsal commands; and
- expected API, module, and console outputs.

They should not receive your `.venv`, editable-install metadata, caches, or `dist/`. If they build the same version, artifact bytes are not guaranteed to be identical across every backend, timestamp, or platform. The foundation claim is narrower: the declared workflow produces an artifact with the expected metadata and behavior. Fully reproducible builds require additional controls.

Have the learner report the base/environment Python versions, wheel filename, wheel digest, installed origin, and all three behavior results. Differences in absolute paths are expected. Differences in distribution version, public result, entry-point output, or artifact contents require investigation before a release is trusted.

8. Key points for installable projects

- `[build-system]`, `[project]`, and `[project.scripts]` are standardized packaging tables; `[tool.hatch...]` is backend-specific configuration.
- Repository, distribution, import-package, module, and console-command names are distinct identities even when some spellings match.
- Runtime dependencies describe what installed product code needs. Build and verification tools belong in development configuration.
- Editable installation supports source development. It is not proof of the wheel another environment receives.
- Module mode and a generated console launcher should delegate to one import-safe callable.
- A build frontend requests artifacts; a backend implements the build hooks.
- Inspect both metadata and file contents. A successful build command alone does not establish artifact completeness.
- The strongest release rehearsal installs the exact wheel into a fresh environment and runs it outside the repository.

References

- PyPA guide: writing `pyproject.toml`
- PyPA tutorial: packaging Python projects
- PyPA build-system interface specification
- PyPA entry-points specification
- PyPA discussion of src and flat layouts
- Python wheel archive format specification
- Python `zipfile` documentation
- Python installed-distribution metadata

Unit Challenge: Release the Moonlight Cipher Kit

Online lesson and runnable notebook

Repair, package, build, inspect, and clean-install a moonlit cipher puzzle whose public API and installed command reveal the same hidden phrase.

- **Level:** Python Foundations · Unit 10 challenge
- **Estimated time:** 60–120 minutes
- **Outcome assessed:** Organize a quiet public package, repair module and console entry points, declare build metadata, isolate its environment, inspect its wheel, and prove the exact artifact works outside the repository.
- **Evidence:** Thirty progressive checks, one debugging record, artifact inspection, and a clean outside-repository run

1. Challenge outcome

The Moonlight Museum has recovered four fragments from a lunar vault. The decoder works, but an overexcited apprentice scattered it across a broken Python package. Importing the package leaks the secret, the advertised command points to a missing callable, module mode produces no useful output, the Python version claim excludes every real visitor, and Git is preparing to preserve generated debris.

Repair and release **Moonlight Cipher Kit 0.1.0**. Both:

```bash
python -m mooncipher
mooncipher
```

must reveal exactly:

```text
OPEN THE LUNAR VAULT
```

An ordinary `import mooncipher` must print nothing. The phrase must come from the package’s public decoding behavior—not from a second hard-coded print added to each launcher. Finally, build a wheel, inspect it, install that exact file in a fresh environment, and repeat the proof from outside the source repository.

Use evidence, not repeated installation guesses

Run one stage at a time. Repair the earliest failed contract, rerun that stage, then continue. Open a hint only after recording the command, observed output, and one hypothesis. :::

2. Understand the acceptance example

The museum’s fragments are:

```python
fragments = [" open-", "_the_", " lunar ", "-vault "]
```

`normalize_fragment(value)` must:

1. treat hyphens and underscores as word separators;
2. remove surrounding and repeated whitespace;
3. return uppercase words separated by one ordinary space; and
4. return `""` when no word remains.

`decode_fragments(values)` normalizes every fragment, discards empty results, and joins the remaining text with one space.

```python
assert normalize_fragment("_open--gate_") == "OPEN GATE"
assert normalize_fragment("__-") == ""
assert decode_fragments(fragments) == "OPEN THE LUNAR VAULT"
assert decode_fragments(["_open_", "", "---", "vault"]) == "OPEN VAULT"
```

The package root must publicly expose both functions:

```python
from mooncipher import decode_fragments, normalize_fragment
```

The callable `main(argv=None)` belongs in `mooncipher.cli`. With no supplied fragments it decodes the museum fixture. With a sequence supplied explicitly, it decodes that sequence. It prints one result and returns `None`; full CLI parsing belongs to Unit 12.

The release metadata contract is:

Field	Required value
distribution name	`moonlight-cipher-kit`
version	`0.1.0`
Python	`>=3.10`
build backend	`hatchling.build`
runtime dependencies	none
console command	`mooncipher`
command target	`mooncipher.cli:main`

3. Start from the contract

Run this bootstrap in a disposable directory. It creates every starter file, including deliberate defects. Do not run it inside another project because it creates a new `moonlight-cipher-kit` folder.

```python
from pathlib import Path
from textwrap import dedent

root = Path("moonlight-cipher-kit")
files = {
    ".gitignore": "*.pyc\n",
```

PYTHON (CONTINUED)

```
    "README.md": dedent(
        """\
        # Moonlight Cipher Kit

        Decode word fragments recovered from the Moonlight Museum's lunar vault.
        """
    ),
    "pyproject.toml": dedent(
        """\
        [build-system]
        requires = ["hatchling>=1.26"]
        build-backend = "hatchling.build"

        [project]
        name = "moonlight-cipher-kit"
        version = "0.1.0"
        description = "Decode fragments from a moonlit puzzle vault."
        readme = "README.md"
        requires-python = ">=99"
        dependencies = []

        [project.scripts]
        mooncipher = "mooncipher.cli:launch"

        [tool.hatch.build.targets.wheel]
        packages = ["src/mooncipher"]
        """
    ),
    "src/mooncipher/decoder.py": dedent(
        """\
        def normalize_fragment(value):
            \"\"\"Return uppercase words separated by one space.\"\"\"
            separated = value.replace("-", " ").replace("_", " ")
            return " ".join(separated.split()).upper()


        def decode_fragments(values):
            \"\"\"Normalize fragments and join the non-empty results.\"\"\"
            cleaned = [normalize_fragment(value) for value in values]
            return " ".join(value for value in cleaned if value)


        print(decode_fragments([" open-", "_the_", " lunar ", "-vault "]))
        """
    ),
    "src/mooncipher/__init__.py": dedent(
        """\
        \"\"\"Tools for decoding the Moonlight Museum's fragments.\"\"\"
        """
    ),
    "src/mooncipher/cli.py": dedent(
        """\
        import sys

        from .decoder import decode_fragments

        DEFAULT_FRAGMENTS = [" open-", "_the_", " lunar ", "-vault "]


        def main(argv=None):
            \"\"\"Decode supplied fragments, or the museum fixture, and print it.\"\"\"
            fragments = list(sys.argv[1:] if argv is None else argv)
            print(decode_fragments(fragments or DEFAULT_FRAGMENTS))


        main()
        """
    ),
```

PYTHON (CONTINUED) (CONTINUED)

```
    "src/mooncipher/__main__.py": dedent(
        """\
        from .cli import main

        main
        """
    ),
    "checks/check_release.py": dedent(
        """\
        import contextlib
        import importlib.util
        import io
        import os
        import subprocess
        import sys
        import tomllib
        from email.parser import BytesParser
        from email.policy import default
        from pathlib import Path
        from tempfile import TemporaryDirectory
        from zipfile import ZipFile

        ROOT = Path(__file__).resolve().parents[1]
        SOURCE = ROOT / "src" / "mooncipher"
        EXPECTED = "OPEN THE LUNAR VAULT"
        checks_run = 0


        def check(condition, message):
            global checks_run
            assert condition, message
            checks_run += 1


        def environment_python(environment):
            relative = Path("Scripts/python.exe") if os.name == "nt" else Path("bin/python")
            return environment / relative


        def command_path(environment, name):
            if os.name == "nt":
                return environment / "Scripts" / f"{name}.exe"
            return environment / "bin" / name


        def run(command, cwd):
            return subprocess.run(
                [str(part) for part in command],
                cwd=cwd,
                text=True,
                capture_output=True,
            )


        def source_checks():
            required = ["__init__.py", "__main__.py", "cli.py", "decoder.py"]
            check(all((SOURCE / name).is_file() for name in required), "source files are missing")

            spec = importlib.util.spec_from_file_location("moon_decoder_check", SOURCE / "decoder.py")
            check(spec is not None and spec.loader is not None, "decoder cannot be loaded")
            module = importlib.util.module_from_spec(spec)
            captured = io.StringIO()
            with contextlib.redirect_stdout(captured):
                spec.loader.exec_module(module)
            check(captured.getvalue() == "", "decoder printed during import")
            check(callable(module.normalize_fragment), "normalize_fragment is missing")
            check(callable(module.decode_fragments), "decode_fragments is missing")
            check(module.normalize_fragment("_open--gate_") == "OPEN GATE", "normalization failed")
```

PYTHON (CONTINUED) (CONTINUED) (CONTINUED)

```
            check(module.normalize_fragment("__-") == "", "empty boundary failed")
            check(
                module.decode_fragments([" open-", "_the_", " lunar ", "-vault "]) == EXPECTED,
                "museum phrase failed",
            )
            check(module.decode_fragments(["_open_", "", "---", "vault"]) == "OPEN VAULT", "empty fragm ↪
    ent filtering failed")


        def metadata_checks():
            data = tomllib.loads((ROOT / "pyproject.toml").read_text(encoding="utf-8"))
            project = data["project"]
            check(project["name"] == "moonlight-cipher-kit", "distribution name is wrong")
            check(project["version"] == "0.1.0", "version is wrong")
            check(project["requires-python"] == ">=3.10", "Python range is wrong")
            check(project["dependencies"] == [], "runtime dependencies must be empty")
            check(data["build-system"]["build-backend"] == "hatchling.build", "backend is wrong")
            check(project["scripts"]["mooncipher"] == "mooncipher.cli:main", "script target is wrong")


        def installed_checks():
            with TemporaryDirectory() as name:
                outside = Path(name)
                api = run(
                    [sys.executable, "-c", "from mooncipher import decode_fragments, normalize_fragment ↪
    ; print(decode_fragments(['open', 'the', 'lunar', 'vault'])); print(normalize_fragment('_moon-gate_ ↪
    '))"],
                    outside,
                )
                check(api.returncode == 0, api.stderr)
                check(api.stdout.splitlines() == [EXPECTED, "MOON GATE"], "public API or quiet import f ↪
    ailed")

                module = run([sys.executable, "-m", "mooncipher"], outside)
                check(module.returncode == 0, module.stderr)
                check(module.stdout.strip() == EXPECTED, "module entry point failed")

                command = run([command_path(Path(sys.prefix), "mooncipher")], outside)
                check(command.returncode == 0, command.stderr)
                check(command.stdout.strip() == EXPECTED, "console command failed")


        def artifact_checks():
            wheels = sorted((ROOT / "dist").glob("*.whl"))
            check(len(wheels) == 1, f"expected one wheel, found {len(wheels)}")
            wheel = wheels[0].resolve()
            with ZipFile(wheel) as archive:
                names = archive.namelist()
                check("mooncipher/decoder.py" in names, "decoder is absent from wheel")
                check("mooncipher/cli.py" in names, "CLI is absent from wheel")
                check(not any("__pycache__" in name or name.startswith("checks/") for name in names), " ↪
    development artifacts entered wheel")
                metadata_name = next(name for name in names if name.endswith(".dist-info/METADATA"))
                metadata = BytesParser(policy=default).parsebytes(archive.read(metadata_name))
                check(metadata["Name"] == "moonlight-cipher-kit" and metadata["Version"] == "0.1.0", "w ↪
    heel metadata is wrong")

            with TemporaryDirectory() as name:
                root = Path(name)
                environment = root / "clean-env"
                outside = root / "outside"
                outside.mkdir()
                subprocess.run([sys.executable, "-m", "venv", str(environment)], check=True)
                python = environment_python(environment)
                install = run([python, "-m", "pip", "install", wheel], outside)
                check(install.returncode == 0, install.stderr)
                proof = run([python, "-m", "mooncipher"], outside)
                check(proof.returncode == 0 and proof.stdout.strip() == EXPECTED, proof.stderr or proof ↪
```

PYTHON (CONTINUED) (CONTINUED) (CONTINUED) (CONTINUED)

```
    .stdout)
                console = run([command_path(environment, "mooncipher")], outside)
                check(console.returncode == 0 and console.stdout.strip() == EXPECTED, console.stderr or ↪
     console.stdout)


        STAGES = {
            "source": source_checks,
            "metadata": metadata_checks,
            "installed": installed_checks,
            "artifact": artifact_checks,
        }

        if __name__ == "__main__":
            stage = sys.argv[1] if len(sys.argv) == 2 else ""
            if stage not in STAGES:
                raise SystemExit("usage: python checks/check_release.py source|metadata|installed|artif ↪
    act")
            STAGES[stage]()
            print(f"{stage}: {checks_run} checks passed")
        """
    ),
}

for relative_path, content in files.items():
    destination = root / relative_path
    destination.parent.mkdir(parents=True, exist_ok=True)
    destination.write_text(content, encoding="utf-8")

print(f"Created {len(files)} files under {root.resolve()}")
```

The starter has seven seeded defects:

1. `decoder.py` prints the secret while being imported;
2. `__init__.py` does not expose the two public functions;
3. `cli.py` launches `main()` during import;
4. `__main__.py` refers to `main` but never calls it behind an execution guard;
5. `requires-python` excludes the course interpreter;
6. the installed script points to nonexistent `launch`; and
7. `.gitignore` omits environments, build output, and caches.

Do not add an eighth workaround such as `sys.path.insert`. Repair the owning contract.

4. Build in small stages

Work from the new `moonlight-cipher-kit` directory.

Stage A: make source imports quiet

Remove import-time action from `decoder.py`. Keep the two function definitions there. The source stage loads that file directly, so it can run before package installation:

BASH

```
python checks/check_release.py source
```

Expected final result for this stage:

```text
source: 9 checks passed
```

Stage B: repair metadata before creating the project environment

Correct `requires-python` and the script target, then run:

```bash
python checks/check_release.py metadata
```

Expected:

```text
metadata: 6 checks passed
```

Stage C: create one development environment and install editable source

```bash
python -m venv .venv
.venv/bin/python -m pip install --editable .
```

On Windows, use `.venv\Scripts\python.exe`. Prove identity before continuing:

```bash
.venv/bin/python -c "import sys; print(sys.executable); print(sys.prefix != sys.base_prefix)"
.venv/bin/python -m pip --version
```

Repair the package facade, remove the unguarded call from `cli.py`, and make `__main__.py` call `main()` only when executed as the top-level module. Then run the installed stage through the environment interpreter:

```bash
.venv/bin/python checks/check_release.py installed
```

Expected:

```text
installed: 6 checks passed
```

Stage D: protect the repository boundary

Expand `.gitignore`, initialize the disposable repository, and inspect before committing:

```bash
git init
git status --short --ignored
git add .gitignore README.md pyproject.toml src checks
git diff --staged
git commit -m "Repair moonlight cipher package"
```

Configure a local course identity first if Git requests one. The staged diff must exclude `.venv/`, `dist/`, `build/`, caches, and installed metadata.

Stage E: build one clean artifact set

Install the build frontend as development tooling, remove stale output, and build:

```bash
.venv/bin/python -m pip install "build>=1"
rm -rf dist build
.venv/bin/python -m build
```

The final artifact stage creates its own second environment and installs the exact wheel. It may take longer than the earlier stages:

```bash
.venv/bin/python checks/check_release.py artifact
```

Expected:

```text
artifact: 8 checks passed
```

Windows learners can remove `dist` and `build` with PowerShell before running the same Python check command.

5. Run progressive assertions

The harness groups **29 automated checks**. Add this final Git inspection as check 30:

```bash
git status --short --ignored
git log --oneline --decorate -n 2
```

Use the list to see progress even before a whole group passes.

Source contract — checks 1–9

- ☐ all four package source files exist;
- ☐ the decoder can be loaded from its file;
- ☐ decoder import prints nothing;
- ☐ `normalize_fragment` is callable;
- ☐ `decode_fragments` is callable;
- ☐ separators and repeated whitespace normalize to `OPEN GATE`;
- ☐ a separator-only fragment becomes empty text;
- ☐ museum fragments decode to `OPEN THE LUNAR VAULT`; and
- ☐ empty normalized fragments do not add extra spaces.

Metadata contract — checks 10–15

- ☐ distribution name is `moonlight-cipher-kit`;
- ☐ version is `0.1.0`;
- ☐ Python requirement is `>=3.10`;
- ☐ runtime dependency list is empty;

- ☐ backend is `hatchling.build`; and
- ☐ script target is `mooncipher.cli:main`.

Development installation — checks 16–21

- ☐ public API imports in a child process outside the repository;
- ☐ public import adds no extra output and both API results are correct;
- ☐ `python -m mooncipher` exits successfully;
- ☐ module mode prints exactly the phrase;
- ☐ installed console launcher exits successfully; and
- ☐ console launcher prints exactly the phrase.

Wheel and clean installation — checks 22–29

- ☐ exactly one wheel is present;
- ☐ wheel contains `mooncipher/decoder.py`;
- ☐ wheel contains `mooncipher/cli.py`;
- ☐ wheel excludes checks and Python caches;
- ☐ wheel metadata contains the expected name and version;
- ☐ exact wheel installs into the fresh environment;
- ☐ clean module mode prints the phrase outside the repository; and
- ☐ clean console command prints the phrase outside the repository.

Repository boundary — check 30

- ☐ Git records only intentional project source; generated/local artifacts are ignored and absent from the commit.

If a group stops on the first assertion, repair it and rerun. The displayed count will advance as earlier checks pass.

6. Use the hint ladder only when needed

Hint 1

There should be exactly one owner for decoding and one owner for command action. `decoder.py` should contain definitions only. The package root imports the two supported functions. Both launch mechanisms should eventually call `mooncipher.cli.main`.

Draw these arrows before editing:

```text
__init__.py  -> decoder.py
cli.py       -> decoder.py
__main__.py  -> cli.py
```

No arrow needs to point back upward.

Hint 2

An imported module should not call its own demo. In `cli.py`, retain the `main(argv=None)` definition but remove the unguarded final call. In `__main__.py`, import `main` and put the call under the same `__name__` guard used in the modules lesson. In `pyproject.toml`, the script target uses `module:attribute` without parentheses.

Hint 3

The package initializer needs explicit relative re-exports and `__all__`. The Python requirement is the course minimum, not a fictional future version. A useful ignore file covers `.venv/`, `__pycache__/`, `*.py[cod]`, `*.egg-info/`, `build/`, `dist/`, and common tool caches. After every metadata or source repair, rerun editable installation before judging its installed entry points.

7. Keep debugging evidence

Preserve one failure that changed your understanding. A useful record has this shape:

Field	Your evidence
Stage and command	for example, `.venv/bin/python checks/check_release.py installed`
Interpreter	output of `sys.executable`
Working directory	absolute project path or temporary outside path
Observed result	return code plus exact stdout/stderr
Contract	what should have been quiet or printed exactly once
Hypothesis	one ownership, metadata, environment, or artifact explanation
Controlled repair	one changed file and why it owns the contract
Regression proof	rerun stage plus one later stage

For example, two copies of the secret phrase suggest two execution paths, not a string-normalization bug. A traceback naming `launch` suggests entry-point metadata, not a missing wheel. A `Requires-Python` refusal points at compatibility metadata before source execution begins.

Do not erase the failed output after repairing it. The contrast between failure and proof is part of the challenge result.

8. Compare with a complete solution

Show every final project file after attempting all stages

`.gitignore`:

GITIGNORE

```
.venv/
__pycache__/
*.py[cod]
*.egg-info/
build/
dist/
.pytest_cache/
```

GITIGNORE (CONTINUED)

```
.mypy_cache/
.ruff_cache/
```

README.md:

MARKDOWN

```
# Moonlight Cipher Kit

Decode word fragments recovered from the Moonlight Museum's lunar vault.

Run `python -m mooncipher` or the installed `mooncipher` command to reveal the
museum fixture.
```

pyproject.toml:

TOML

```
[build-system]
requires = ["hatchling>=1.26"]
build-backend = "hatchling.build"

[project]
name = "moonlight-cipher-kit"
version = "0.1.0"
description = "Decode fragments from a moonlit puzzle vault."
readme = "README.md"
requires-python = ">=3.10"
dependencies = []

[project.scripts]
mooncipher = "mooncipher.cli:main"

[tool.hatch.build.targets.wheel]
packages = ["src/mooncipher"]
```

src/mooncipher/decoder.py:

PYTHON

```
def normalize_fragment(value):
    """Return uppercase words separated by one space."""
    separated = value.replace("-", " ").replace("_", " ")
    return " ".join(separated.split()).upper()


def decode_fragments(values):
    """Normalize fragments and join the non-empty results."""
    cleaned = [normalize_fragment(value) for value in values]
    return " ".join(value for value in cleaned if value)
```

src/mooncipher/__init__.py:

PYTHON

```
"""Tools for decoding the Moonlight Museum's fragments."""

from .decoder import decode_fragments, normalize_fragment

__all__ = ["decode_fragments", "normalize_fragment"]
```

src/mooncipher/cli.py:

```python
import sys

from .decoder import decode_fragments

DEFAULT_FRAGMENTS = [" open-", "_the_", " lunar ", "-vault "]


def main(argv=None):
    """Decode supplied fragments, or the museum fixture, and print it."""
    fragments = list(sys.argv[1:] if argv is None else argv)
    print(decode_fragments(fragments or DEFAULT_FRAGMENTS))
```

`src/mooncipher/__main__.py`:

```python
from .cli import main


if __name__ == "__main__":
    main()
```

Keep the starter `checks/check_release.py` unchanged. It is development support, not part of the import package. Run all four stages again from clean source and verify the staged Git diff excludes generated paths.

9. Adapt to a changed museum rule

The museum finds a fifth fragment, `_TONIGHT_`. Prepare release `0.2.0` so the default phrase becomes:

```text
OPEN THE LUNAR VAULT TONIGHT
```

Requirements:

1. change the version in project metadata;
2. extend `DEFAULT_FRAGMENTS` rather than hard-coding a new final phrase;
3. keep `decode_fragments` and its original assertions compatible;
4. update the expected fixture in the check harness;
5. remove old `dist/` artifacts before rebuilding;
6. inspect wheel metadata for `0.2.0`; and
7. clean-install the new exact wheel and prove both entry routes outside the repository.

Explain why changing only `EXPECTED` in the harness would manufacture a failure rather than implement the new product rule. Then create one focused Git commit whose staged diff contains the version, fixture, and expectation changes but no wheel or environment.

10. Check your understanding

Moonlight Cipher release check

Connect each answer to the package, environment, and artifact evidence.

1. Where should the unconditional demo print be removed first?

- **A.** Only from the README
- **B.** Only from the check harness
- **C.** From `decoder.py`, because importing reusable definitions must be quiet
- **D.** From the wheel filename

2. Which script target matches the repaired callable?

- **A.** `src/mooncipher/cli.py()`
- **B.** `moonlight-cipher-kit:main`
- **C.** `mooncipher.cli.launch`
- **D.** `mooncipher.cli:main`

3. What is the strongest evidence that the wheel—not repository source—reveals the phrase?

- **A.** Install the exact wheel in a fresh environment and run from an unrelated directory.
- **B.** Run the editable package from the repository root.
- **C.** Read only the source TOML.
- **D.** Keep several old wheels and choose one arbitrarily.

4. Which path belongs in the commit for this challenge?

- **A.** `.venv/`
- **B.** `src/mooncipher/decoder.py`
- **C.** `dist/*.whl`
- **D.** `__pycache__/`

5. How should release 0.2.0 add `TONIGHT` while preserving the decoder contract?

- **A.** Replace `decode_fragments` with one hard-coded phrase.
- **B.** Edit only the expected assertion.
- **C.** Extend the default fragment fixture, bump metadata, rebuild, and repeat clean verification.
- **D.** Commit the old virtual environment.

Write your choices before checking the answer key at the end of this book.

11. Decide whether the challenge is complete

Record completion only after:

- source, metadata, installed, and artifact stages all pass;
- check 30 confirms generated paths stayed out of Git;
- imports are quiet and both entry routes print exactly one phrase;
- the wheel’s package files and metadata were inspected;
- the clean installation ran outside the repository;
- one debugging record contains before-and-after evidence; and

- the `0.2.0` changed rule also passes from a rebuilt, clean-installed wheel.

This button records progress only in this browser. It does not submit work, grade it, verify identity, or issue a certificate.

Key points

- Reusable modules define behavior without launching demonstrations at import.
- Package facades and both entry routes should point toward one-directional ownership, not duplicate the secret phrase.
- Metadata errors can prevent installation before product source runs; diagnose the layer named by the evidence.
- Environment Python, module origin, wheel contents, and installed metadata are stronger proof than a prompt or successful build message.
- A fresh environment outside the repository tests the artifact a user actually receives.
- Git should preserve deliberate source and configuration, not disposable environments, caches, or local build output.

Milestone Challenge: Recover the Observatory Log

Online lesson and runnable notebook

Build a reproducible package that parses, validates, and summarizes a partly damaged CSV log.

- **Goal:** Build a reproducible package that parses, validates, and summarizes a partly damaged CSV log.
- **Estimated time:** 2–4 hours of active work.
- **Before starting:** Complete the milestone units and their unit challenges.
- **Keep:** Your source, observed outputs, checks, and a debugging note.

This community challenge is ungraded. Use the checks to review your own work; there is no submission service or automatic grade. Completion is self-reported in this browser on this device.

1. Recover the observatory log

Create load_readings(path) and a python -m entry point. The CSV header is station,reading. Strip surrounding spaces; accept nonempty stations and finite float readings. Skip invalid rows and return row-numbered explanations alongside accepted records. Missing files, empty files, and wrong headers are fatal errors with clear messages. Summarize accepted readings by station with count and mean. Use the standard library.

Predict an ordinary result before coding. Start with the smallest part you can run and inspect. Use Colab for notebooks and the local workspace introduced in the units for packaging or command-line tasks.

2. Set up the experiment

PYTHON

```
sample_csv = """station,reading
A,10
B,20
A,14
B,bad
,8
A,nan
"""
# Save readings.csv, then implement parsing, validation, and reporting.
```

The starter is deliberately incomplete. Implement the contract; do not change the expected results to match an incorrect implementation.

3. Check the observable results

1. A has count 2 and mean 12.0; B has count 1 and mean 20.0.
2. Rows 5, 6, and 7 are rejected with distinct reasons; the header is row 1.
3. A missing file or wrong header produces a clear fatal error.

4 A valid header without data produces an empty summary and no rejected records.

5 Importing the package does not read a file or print. Document a command that runs in a clean environment.

Record actual outputs beside predictions. Reduce a failure to a small example, change one cause at a time, and repeat the failed check after repairing it.

4. Change one condition

Add a station containing a comma using a quoted CSV field, and a reading of infinity. Preserve the station name but reject infinity. Explain why splitting lines on commas fails.

5. Review a hint after attempting the task

Hint: connect the relevant tools

Use csv.DictReader, enumerate starting at 2, float conversion, and math.isfinite. Separate parsing, validation, grouping, and presentation; catch expected errors at the input boundary.

6. Explain and self-review

Include the fixture, output, accepted/rejected counts, assertions, and run instructions. Explain a row-level error versus a file-level error, and save a minimal reproduction of one repaired bug.

Review path: compare your result with the contract

Check the ordinary result first, then the boundary cases and changed condition. Explain which rule justifies each result. An implementation that differs from the hint can still satisfy the contract; explain why yours does.

Check the reasoning behind your result

Choose an answer, then review the explanation.

1. What should happen with a wrong CSV header?

- **A.** Convert every value to zero.
- **B.** Silently skip all rows.
- **C.** Stop with a clear file-level error.
- **D.** Treat it as a station.

Write your choices before checking the answer key at the end of this book.

7. Save your work and record completion

Save the notebook or project somewhere you control. Include one decision, one failed check, its repair, and a result you can reproduce. Record completion after reviewing the checks; you can undo it later.

Return to the milestone overview.

Ship a Tested and Documented Python Tool

Online lesson and runnable notebook

Deliver a tested, installable command-line tool with documented behavior and reproducible checks.

What you will build

Deliver a tested, installable command-line tool with documented behavior and reproducible checks.

Follow the units in order and complete each unit challenge. Keep examples and debugging notes that help explain your choices. This overview is orientation, not a scored activity.

Unit sequence

1. Object-Oriented Python and Dataclasses — 5 lessons and a unit challenge.
2. Command-Line Applications — 5 lessons and a unit challenge.
3. Testing Python Programs — 7 lessons and a unit challenge.
4. Code Quality and Maintainability — 7 lessons and a unit challenge.
5. Documentation as Part of the Product — 7 lessons and a unit challenge.

Prepare and pace your work

Complete the preceding milestone in the course sequence, or review its outcomes if you already have that experience. Use each unit’s active-time estimate and pause at lesson boundaries. Plan another 2–4 hours for the milestone challenge, including rerunning checks and explaining your result.

Use Colab for notebook exercises and the workspace introduced in the units for local environment, packaging, and command-line tasks.

Connect the units

Finish with Publish a Tiny Community Library Tool. Combine the milestone skills and check a changed condition. The challenge provides a contract, expected results, a hint, and self-review guidance.

Community learning is ungraded. Completion controls record your own progress only in this browser on this device; they do not issue a verified credential.

Begin the first unit.

Unit 11: Object-Oriented Python and Dataclasses

Online lesson and runnable notebook

Learn when an object makes a program clearer, how classes protect evolving state, and how small collaborating objects form a dependable model.

- **Level:** Python Foundations
- **Estimated time:** 45–60 minutes
- **Unit outcome:** Choose between functions, collections, dataclasses, and stateful classes; build objects that preserve valid state; explain instance attributes and bound methods; compose replaceable collaborators; and use inheritance only when a subtype can honor the same behavioral contract.
- **Practice in:** Google Colab, JupyterLab, or a local Python interpreter

Imagine that you are tracking a hiking expedition. On the first morning, two values may be enough:

```python
expedition_name = "Dawn Trail"
distance_km = 0
```

Soon you add a list of checkpoints, a maximum distance, and functions that record progress. Then a second expedition begins. Every function must receive the right dictionary, every dictionary must contain the right keys, and every update must respect the same rules. The code can still work, but the program now has a useful question to answer:

Which part of the program owns this state and keeps it valid as it changes?

A class is one possible answer. It creates a new kind of object whose data and allowed operations can be understood together. It is not automatically the best answer, and this unit will not ask you to turn every noun into a class. Instead, you will learn to choose the smallest structure that makes the rules clear.

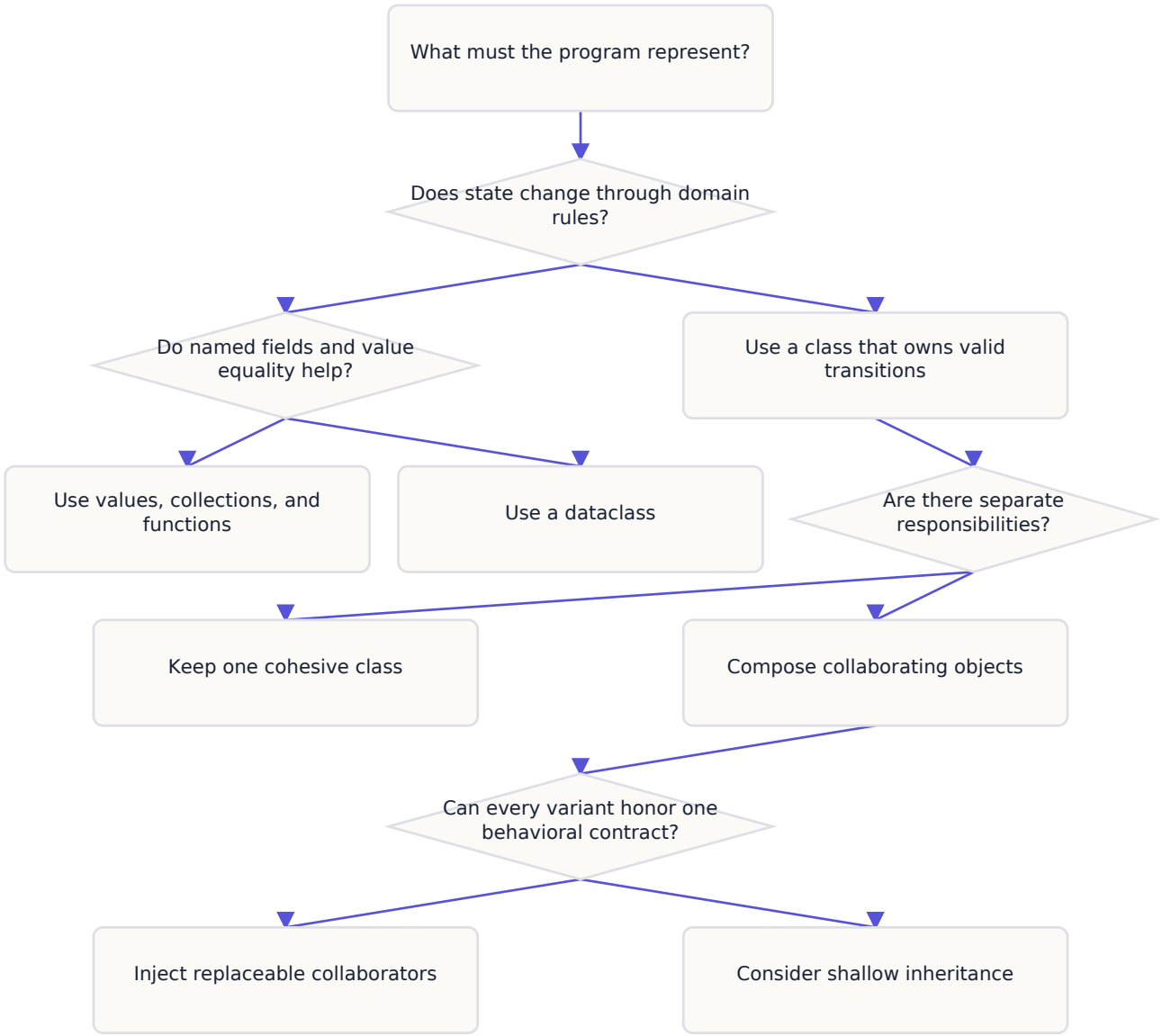
1. Choose an abstraction because it earns its place

Python gives you several ways to organize a solution. They overlap, so the decision begins with the behavior you need rather than a slogan:

Need	Good first choice	Why
one value	string, number, tuple, or another value	no extra structure is needed
passive named data	dictionary or small dataclass	fields travel together
a transformation with no owned state	function	inputs and output make the contract visible

Need	Good first choice	Why
a value-like record with representation and equality	dataclass	Python can generate repetitive field-based methods
evolving state that must obey rules	regular class	one object can own valid transitions
several responsibilities that cooperate	composed objects	each object can own one coherent part
interchangeable specialized behavior	composition first; inheritance only for an honest subtype	callers depend on behavior rather than implementation

This decision ladder has an intentional path back to simple code. More abstraction is not the definition of better design.



Does state change through domain rules? → Do named fields and value equality help?: No

Do named fields and value equality help? → Use values, collections, and functions: No

Do named fields and value equality help? → Use a dataclass: Yes

Does state change through domain rules? → Use a class that owns valid transitions: Yes

Are there separate responsibilities? → Keep one cohesive class: No

Are there separate responsibilities? → Compose collaborating objects: Yes

Can every variant honor one behavioral contract? → Inject replaceable collaborators: No

Can every variant honor one behavioral contract? → Consider shallow inheritance: Yes

The arrows are questions, not laws. A dictionary and functions can be an excellent final design. A dataclass can contain methods. A regular class can be value-like. The point is to make a conscious choice and explain its trade-offs.

2. Connect new syntax to Python you already know

Classes do not replace the ideas from earlier units. They combine them:

- **Names and assignment:** `self.distance = 3` binds a name in one instance's namespace.
- **Functions:** a method begins as a function defined in a class body. Accessing it through an instance binds that instance to the call.
- **Identity and mutation:** two equal-looking instances can be different objects, and mutable instance state can be shared accidentally if it is stored in the wrong place.
- **Exceptions:** constructors and methods raise precise errors when an operation would violate a rule.
- **Modules:** a class belongs in the module that owns its responsibility; it is not a reason to put an entire project in one file.

Unit 6 prepared you to reason about identity, aliases, copying, and hashability. Unit 8 prepared you to define failure contracts. Unit 10 prepared you to separate public interfaces from implementation files. This unit applies those skills to objects.

We will use plain assertions to check behavior. Unit 13 later introduces pytest, fixtures, parametrization, and test doubles. For now, a short assertion keeps attention on the object's contract:

PYTHON
```
assert tracker.distance_km == 5
```

3. Follow one idea from a record to a system

The lessons build in five deliberate steps:

Lesson	Design question	Evidence you will produce
1. Build Your First Useful Class	When does grouping state and behavior improve a working dictionary design?	independent instances, inspected namespaces, bound-method traces, and a repaired shared-state bug
2. Keep Objects in Valid States	How can every construction and update path preserve the same rules?	atomic methods, useful properties, an alternate constructor, and failed operations that leave state unchanged

Lesson	Design question	Evidence you will produce
3. Use Dataclasses for Value-Like Objects	Which field-based behavior should Python generate, and what must you still design?	safe defaults, meaningful representation/equality, post-initialization validation, and frozen-value updates
4. Choose Composition Before Inheritance	How can objects cooperate without a brittle hierarchy?	injected collaborators, delegation, duck-typed substitution, one honest subtype, and one rejected inheritance design
5. Model a Complete Road-Crossing Game	How do value objects, entities, policies, and an orchestrator produce one deterministic turn?	acceptance examples, an object graph, a full turn result, and a changed rule placed in the correct owner

Focused examples use expeditions, batteries, map tiles, and museum tours. The last lesson combines the ideas in a road-crossing game model. It deliberately keeps keyboard input, drawing, timing, and randomness outside the rules so that you can predict and verify every turn.

4. Learn four roles without forcing four base classes

The unit uses a small vocabulary for responsibilities:

Value object

Represents a value whose fields determine what it means. Coordinates and move descriptions are good candidates. Frozen dataclasses often fit.

Stateful entity

Has an identity and changes over time while preserving rules. A battery or player can own methods that perform valid transitions.

Collaborator

Provides behavior another object uses. A route planner can be replaced by a different planner if both understand the same request.

Orchestrator

Coordinates a small workflow without taking over every responsibility. A game can ask a player to move, lanes to advance, and rules to report the result.

These are reasoning labels, not mandatory parent classes. Creating `ValueObjectBase`, `EntityBase`, and `ManagerBase` would add machinery before it adds understanding. First make the responsibility visible in ordinary Python.

5. Expect to investigate bad designs, not only polished ones

Premium practice includes mistakes that real programs make:

- a mutable list stored on a class and unexpectedly shared by every instance;
- a method missing `self` and receiving a surprising positional argument;
- a constructor that accepts impossible state;

- an update that changes one field before discovering another input is invalid;
- a dataclass field whose default list is shared or rejected;
- a frozen dataclass that still contains a mutable list;
- an object that creates its own random collaborator and cannot be checked deterministically;
- a subclass that refuses an operation its base type promises;
- a game method that prints, reads input, mutates global state, and applies rules in one inseparable block.

For each mistake, you will observe the error or incorrect state, form one hypothesis, repair the owner of the rule, and rerun a focused check. The repair matters, but the explanation is the transferable skill.

6. Finish with a different world

The unit challenge asks you to run a tiny robot tournament. It does not reuse the road-crossing classes. You will create immutable moves and turn results, robots that protect energy and shield values, and an arena that alternates accepted turns. Progressive assertions reveal one contract at a time, while three hints provide increasingly direct support.

The final rule change adds a combo spotlight for a repeated move. That change tests whether history belongs to the arena, whether rejected turns stay atomic, and whether the existing classes can evolve without a rewrite.

Plan **24–34 hours** for this unit, including typing examples, making predictions, completing checkpoints, debugging, and building the challenge. Useful stopping points are the end of each numbered section or checkpoint. The challenge itself is designed for approximately 60–120 focused minutes after the five lessons.

7. Evidence that you are ready to continue

By the end, you should be able to show and explain:

- two instances whose changes remain independent;
- the difference between an instance attribute and an intentional class attribute;
- where `self` comes from in an instance method call;
- an invariant and every public operation that can affect it;
- why one interface is a property and another is a method;
- exactly which methods a dataclass generated;
- why `default_factory` prevents shared state;
- why `frozen=True` is not deep immutability;
- one replaceable collaborator used through behavior rather than a shared base class;
- one inheritance relationship that passes a behavioral substitution test;
- a deterministic game turn whose state changes and result can be asserted; and
- a class you deliberately chose **not** to create.

That last item is important. Object-oriented design is successful when the program’s responsibilities become easier to see and change—not when the class count increases.

References

- Python tutorial: Classes
- Python data model
- Python `dataclasses`
- CS50P: Object-Oriented Programming
- University of Helsinki Python Programming MOOC: Objects and methods

Build Your First Useful Class

Online lesson and runnable notebook

Refactor a working dictionary into a class, create independent instances, inspect instance and class state, and trace how Python binds self to method calls.

- **Level:** Python Foundations
- **Estimated time:** 4–6 hours
- **You will learn:** Compare a dictionary-plus-functions design with a class, create independent instances, distinguish instance and class attributes, and trace where `self` comes from in a bound method call.
- **Practice in:** Google Colab, JupyterLab, or a local Python interpreter

An expedition team records its trail name, distance, and visited checkpoints. A dictionary and functions can solve that task well. We will begin there—not because the code is wrong, but because a useful class should improve a real design rather than appear only to demonstrate syntax.

As you work, answer these questions:

- What becomes easier when one object owns both state and valid operations?
- Where does Python store an instance's attributes?
- Why do two instances of the same class change independently?
- How does `tracker.record_distance(3)` supply `self`?
- When does a class attribute help, and when does it accidentally share state?

1. Start with a dictionary that works

The first version uses a dictionary for state and functions for behavior:

PYTHON

```
def create_expedition(name):
    """Return a new expedition record."""
    return {
        "name": name,
        "distance_km": 0.0,
        "checkpoints": [],
    }


def record_distance(expedition, distance_km):
    """Add a non-negative distance to an expedition."""
    if distance_km < 0:
        raise ValueError("distance_km cannot be negative")
    expedition["distance_km"] += distance_km


def visit_checkpoint(expedition, checkpoint):
    """Record a non-empty checkpoint name."""
    cleaned = checkpoint.strip()
    if not cleaned:
        raise ValueError("checkpoint cannot be blank")
    expedition["checkpoints"].append(cleaned)
```

Create two expeditions and use the functions:

```python
dawn = create_expedition("Dawn Trail")
dusk = create_expedition("Dusk Trail")

record_distance(dawn, 3.5)
visit_checkpoint(dawn, "River Gate")

print(dawn)
print(dusk)
```

The output shows that the dictionaries are independent:

```text
{'name': 'Dawn Trail', 'distance_km': 3.5, 'checkpoints': ['River Gate']}
{'name': 'Dusk Trail', 'distance_km': 0.0, 'checkpoints': []}
```

This is valid procedural Python. Keep it when the data shape is small, the operations are few, and passing a record explicitly remains clear.

Now imagine ten functions spread across several modules. Every caller must know the exact key spelling. Any code can assign `dawn["distance_km"] = -900`. A checkpoint function could accidentally receive a customer dictionary with similar keys. The design lacks a clear owner, even though every individual line is legal.

A class is not automatically shorter

The class version may contain more lines. Its value is a named kind of object, a clear call interface, and one place to keep related rules—not line-count reduction. :::

2. Move cohesive state and behavior together

Define a class with the `class` statement:

```python
class ExpeditionTracker:
    """Track the progress of one expedition."""

    def __init__(self, name):
        self.name = name
        self.distance_km = 0.0
        self.checkpoints = []

    def record_distance(self, distance_km):
        if distance_km < 0:
            raise ValueError("distance_km cannot be negative")
        self.distance_km += distance_km

    def visit(self, checkpoint):
        cleaned = checkpoint.strip()
        if not cleaned:
            raise ValueError("checkpoint cannot be blank")
        self.checkpoints.append(cleaned)
```

Read the definition from the outside inward:

1 `class ExpeditionTracker:` asks Python to execute a class body and bind the resulting class object to the name `ExpeditionTracker`.

2 Each `def` creates a function in the class namespace.

3 `__init__` describes how to initialize one new instance.

4 Assignments such as `self.name = name` store data on that instance.

5 Later methods read or change the instance received as `self`.

Calling the class produces instances:

```python
dawn = ExpeditionTracker("Dawn Trail")
dusk = ExpeditionTracker("Dusk Trail")

dawn.record_distance(3.5)
dawn.visit("River Gate")
dusk.record_distance(1.25)

print(dawn.name, dawn.distance_km, dawn.checkpoints)
print(dusk.name, dusk.distance_km, dusk.checkpoints)
```

```text
Dawn Trail 3.5 ['River Gate']
Dusk Trail 1.25 []
```

`dawn` and `dusk` are both `ExpeditionTracker` instances, but they are not the same object. Each call created a separate instance and `__init__` assigned a new checkpoint list to it.

```python
print(dawn is dusk)
print(type(dawn).__name__)
print(isinstance(dawn, ExpeditionTracker))
```

```text
False
ExpeditionTracker
True
```

`type(dawn)` identifies the class that created the instance. `isinstance(dawn, ExpeditionTracker)` asks whether `dawn` is an instance of that class or one of its subclasses. The `is` expression asks the stricter identity question from Unit 6.

Checkpoint: instances own independent state

Trace class calls and independent instances

Choose the result supported by the class definition and calls.

1. `dawn` **and** `dusk` **are created by two separate** `ExpeditionTracker(...)` **calls. After** `dawn.visit("Gate")`**, what is** `dusk.checkpoints`**?**

- **A.** `[]`
- **B.** `["Gate"]`
- **C.** The class name
- **D.** A `TypeError`

2. Which expression asks whether `dawn` **and** `dusk` **are the exact same object?**

- **A.** `dawn == dusk`

- **B.** `dawn is dusk`
- **C.** `type(dawn)`
- **D.** `isinstance(dawn, ExpeditionTracker)`

3. What does executing a `class ExpeditionTracker:` statement bind to the name `ExpeditionTracker`?

- **A.** The first expedition dictionary
- **B.** One hidden global instance
- **C.** A class object that can be called to produce instances
- **D.** Only the text of the class body

Write your choices before checking the answer key at the end of this book.

3. See what initialization does—and does not do

It is common to say "`__init__` creates the object," but that shortcut causes confusion later. A more accurate beginner model is:

1. calling `ExpeditionTracker("Dawn Trail")` starts Python's instance construction machinery;
2. Python obtains a new instance;
3. Python passes that instance to `__init__` as `self` along with the supplied arguments; and
4. the class call returns the initialized instance.

`__new__` participates in creating the instance, but customizing it is outside this foundation unit. The important correction is that `__init__` initializes an instance and must return `None`.

This mistake is legal to write but fails when the class is called:

```python
class BrokenTracker:
    def __init__(self, name):
        self.name = name
        return self


try:
    BrokenTracker("Dawn Trail")
except TypeError as error:
    print(type(error).__name__)
    print(error)
```

Python reports that `__init__()` should return `None`, not a `BrokenTracker`. Remove the `return self`. The class call already returns the instance.

Inspect the instance namespace

Most ordinary instances keep their directly assigned attributes in a dictionary-like namespace. `vars` lets you inspect it:

```python
tracker = ExpeditionTracker("Cloud Path")
print(vars(tracker))
```

PYTHON (CONTINUED)

```
tracker.record_distance(2)
tracker.visit("Old Bridge")
print(vars(tracker))
```

TEXT

```
{'name': 'Cloud Path', 'distance_km': 0.0, 'checkpoints': []}
{'name': 'Cloud Path', 'distance_km': 2.0, 'checkpoints': ['Old Bridge']}
```

`vars(tracker)` is powerful diagnostic evidence, but normal callers should use the object's public interface. Reaching into `vars` to bypass methods would defeat the rules those methods own.

The class has a different namespace:

PYTHON

```
print("record_distance" in vars(ExpeditionTracker))
print("distance_km" in vars(ExpeditionTracker))
print("distance_km" in vars(tracker))
```

The method name is defined on the class. `distance_km` is assigned to this instance by `__init__`. That distinction prepares us to understand method binding.

Catch misspelled and surprise attributes

When lookup cannot find a requested name, Python raises `AttributeError`:

PYTHON

```
tracker = ExpeditionTracker("Cloud Path")

try:
    print(tracker.distnace_km)
except AttributeError as error:
    print(type(error).__name__)
    print(error)
```

Read the instance type and missing name in the message. Here, `distnace_km` is a typo; installing a package or recreating the notebook will not fix it. Compare the request with `vars(tracker)` and the class's documented interface.

Ordinary Python instances also allow a caller to create a new attribute by assignment:

PYTHON

```
tracker.distnace_km = 900

print(tracker.distnace_km)
print(tracker.distance_km)
print(vars(tracker))
```

The typo now creates a second field while the real distance remains `0.0`. That flexibility is useful in exploratory Python, but it means naming discipline and focused checks matter. Later tools can restrict attributes or catch names statically, but this foundation course first makes the runtime behavior visible.

Class bodies execute when Python reaches the class statement. Avoid placing input, network access, or demonstrations there. This small example prints while the class is being defined, before any instance exists:

```python
class NoisyDefinition:
    print("defining NoisyDefinition")

    def __init__(self, value):
        self.value = value
```

Method bodies do not run during class definition; `print` is directly in the class body, so it does. Real class bodies normally contain method definitions, documented shared settings, and small declarative expressions—not program launch behavior.

4. Watch Python bind a method to one instance

Access the method through the class and through an instance:

```python
tracker = ExpeditionTracker("Cloud Path")

print(ExpeditionTracker.record_distance)
print(tracker.record_distance)
```

The exact addresses vary, but the first representation identifies a function and the second identifies a **bound method**. The bound method remembers both the underlying function and the instance:

```python
bound = tracker.record_distance

print(bound.__self__ is tracker)
print(bound.__func__ is ExpeditionTracker.record_distance)
```

Both results are `True`.

These two calls have the same effect:

```python
tracker.record_distance(2.5)
ExpeditionTracker.record_distance(tracker, 1.5)

print(tracker.distance_km)
```

```text
4.0
```

In `tracker.record_distance(2.5)`, Python obtains a bound method and supplies `tracker` as the first argument. The explicit form calls the underlying function through the class and supplies the instance manually. Prefer the first form in ordinary code; the second form is useful evidence for understanding `self`.

`self` is not a keyword. It is the strong Python naming convention for the first parameter of an instance method. Calling it `this_object` would run, but would surprise every Python reader.

Diagnose a missing `self`

Consider this definition:

```python
class Badge:
    def label(prefix):
        return f"{prefix}: explorer"
```

The syntax is valid. The problem appears at the call:

```python
badge = Badge()

try:
    print(badge.label("Lead"))
except TypeError as error:
    print(type(error).__name__)
    print(error)
```

Python binds `badge` to the first parameter named `prefix`, then the explicit `"Lead"` becomes a second positional argument. The function only declares one, so Python reports that two were given.

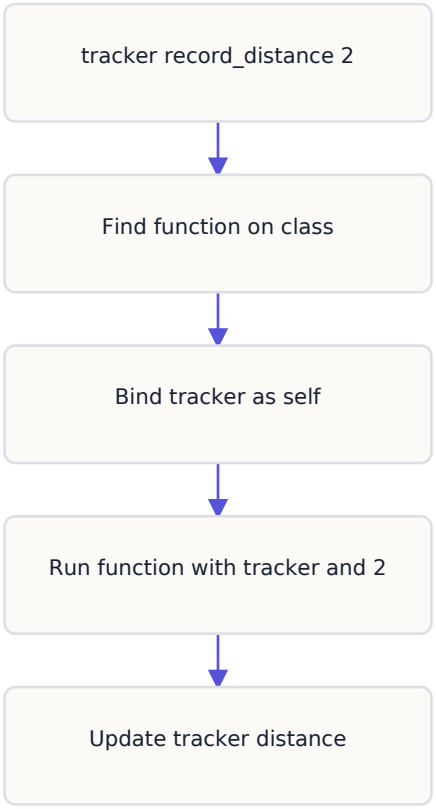
Repair the signature:

```python
class Badge:
    def label(self, prefix):
        return f"{prefix}: explorer"


print(Badge().label("Lead"))
```

The fix is not “add `self` because methods always need magic.” It is “declare a parameter to receive the instance Python binds to this method call.”

An instance method lives on the class. Accessing it through an instance creates a bound method for that access.



Checkpoint: trace `self` and initialization

Follow an instance method call

Choose the explanation that matches Python's observable behavior.

1. What does `tracker.record_distance(2)` supply as `self`?

- **A.** The number `2`
- **B.** The class name as text
- **C.** A new hidden tracker
- **D.** The existing object referenced by `tracker`

2. Which statement describes `__init__` accurately for this lesson?

- **A.** It initializes an instance obtained by Python's construction machinery.
- **B.** It must return the new instance.
- **C.** It runs only when an attribute is missing.
- **D.** It is a class attribute shared as mutable state.

3. Why can `badge.label("Lead")` report two arguments for `def label(prefix)`?

- **A.** Strings always count twice.
- **B.** Python binds `badge` to the first parameter, then also supplies the explicit string.
- **C.** Every method receives its class name twice.

- **D.** The return value adds an argument.

Write your choices before checking the answer key at the end of this book.

5. Put shared settings on the class—not shared personal state

A class attribute is assigned in the class body rather than to `self`:

```python
class ExpeditionTracker:
    distance_unit = "km"

    def __init__(self, name):
        self.name = name
        self.distance_km = 0.0
        self.checkpoints = []

    def record_distance(self, distance_km):
        if distance_km < 0:
            raise ValueError("distance_km cannot be negative")
        self.distance_km += distance_km
```

The unit label is intentionally shared:

```python
dawn = ExpeditionTracker("Dawn")
dusk = ExpeditionTracker("Dusk")

print(dawn.distance_unit)
print(dusk.distance_unit)
print("distance_unit" in vars(dawn))
print("distance_unit" in vars(ExpeditionTracker))
```

Python does not find `distance_unit` directly on `dawn`, so it finds the name on the class. This common lookup model explains why both instances see `"km"`.

Assigning through one instance creates or changes an instance attribute; it does not rewrite the class attribute:

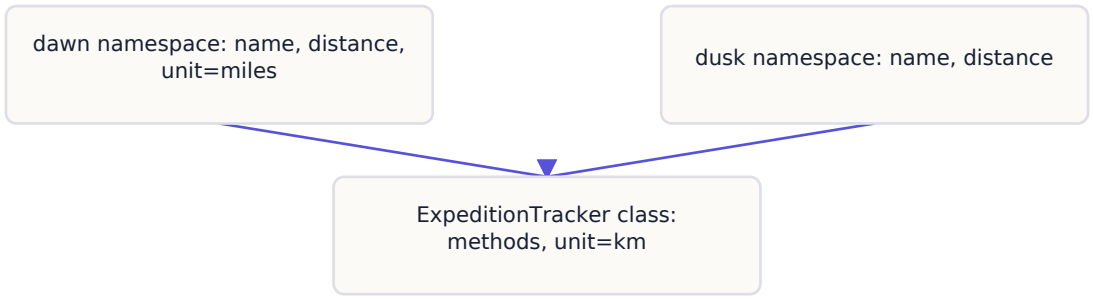
```python
dawn.distance_unit = "miles"

print(dawn.distance_unit)
print(dusk.distance_unit)
print(ExpeditionTracker.distance_unit)
print(vars(dawn))
```

`dawn` now has its own `distance_unit` that **shadows** the class value. `dusk` still finds `"km"` on the class. Deleting `dawn.distance_unit` would reveal the class value again.

This is a useful simplified model for normal attributes. Properties and other managed attributes add lookup behavior, and inheritance adds base classes. We will observe properties next rather than pretending this simplified order is a complete implementation algorithm.

Each instance owns its direct state and refers to the same class for methods and shared settings.



Repair the shared-list trap

Mutable class attributes often create an accidental connection:

```python
class BrokenExpedition:
    checkpoints = []

    def __init__(self, name):
        self.name = name

    def visit(self, checkpoint):
        self.checkpoints.append(checkpoint)
```

Predict the second output before running it:

```python
north = BrokenExpedition("North")
south = BrokenExpedition("South")

north.visit("Pine Gate")

print(north.checkpoints)
print(south.checkpoints)
print(north.checkpoints is south.checkpoints)
```

Both instances find the same list on the class, so both print `['Pine Gate']`, and the identity check is `True`.

Repair the owner of the state:

```python
class Expedition:
    def __init__(self, name):
        self.name = name
        self.checkpoints = []

    def visit(self, checkpoint):
        self.checkpoints.append(checkpoint)
```

Now each `__init__` call creates a fresh list and assigns it to that instance. Use class attributes for genuinely shared facts or behavior—course-wide units, protocol versions, fixed limits—not for each object's evolving collection.

6. Notice the difference between returning and changing

Methods are still functions. They can return values, cause side effects, or do both. Make that contract deliberate:

```python
class DistanceLog:
    def __init__(self):
        self._entries = []

    def record(self, distance_km):
        """Store a distance and return the new total."""
        if distance_km < 0:
            raise ValueError("distance_km cannot be negative")
        self._entries.append(distance_km)
        return sum(self._entries)

    def total(self):
        """Return the total without changing the log."""
        return sum(self._entries)
```

`record` changes `_entries` and reports the new total. `total` only answers a question. A caller can verify both parts:

```python
log = DistanceLog()

new_total = log.record(2.5)
same_total = log.total()

assert new_total == 2.5
assert same_total == 2.5
```

Do not return `self` by habit just to chain calls. Fluent chains can obscure when mutation happens, especially for beginners. Return the result the caller needs, or return `None` when the action itself is the complete contract.

7. Build a trail counter and explain why it is a class

Create `TrailCounter` with these names and contracts:

```python
class TrailCounter:
    distance_unit = "km"

    def __init__(self, trail_name):
        """Start one named trail with zero distance and no checkpoints."""
        ...

    def add_distance(self, amount):
        """Add a non-negative amount and return the new total."""
        ...

    def reach(self, checkpoint):
        """Store a non-blank checkpoint and return its one-based number."""
        ...

    def summary(self):
        """Return `<trail>: <distance> km, <count> checkpoints`."""
        ...
```

Your implementation must satisfy:

```python
ridge = TrailCounter("Ridge Run")
marsh = TrailCounter("Marsh Walk")

assert ridge.add_distance(2.5) == 2.5
assert ridge.add_distance(1.0) == 3.5
assert ridge.reach("Stone Arch") == 1
assert ridge.reach("North Lookout") == 2
assert ridge.summary() == "Ridge Run: 3.5 km, 2 checkpoints"

assert marsh.summary() == "Marsh Walk: 0.0 km, 0 checkpoints"
assert ridge.checkpoints is not marsh.checkpoints
assert TrailCounter.distance_unit == "km"

try:
    ridge.add_distance(-1)
except ValueError as error:
    assert str(error) == "amount cannot be negative"
else:
    raise AssertionError("negative distance should fail")

try:
    ridge.reach("   ")
except ValueError as error:
    assert str(error) == "checkpoint cannot be blank"
else:
    raise AssertionError("blank checkpoint should fail")
```

After it passes, write four sentences:

1. Which attributes belong to each instance?
2. Why is `distance_unit` a class attribute?
3. What instance becomes `self` in `ridge.reach("Stone Arch")`?
4. What does the class improve compared with a dictionary and functions?

Hint 1

In `__init__`, assign `trail_name`, `distance_km`, and a newly created `checkpoints` list to `self`. Do not place the list in the class body.

Hint 2

Validate before assigning or appending. `reach` can strip the supplied text, append the cleaned value, and return `len(self.checkpoints)`.

Hint 3

`summary` only reads state. An f-string can use `self.trail_name`, `self.distance_km`, `len(self.checkpoints)`, and `self.distance_unit`.

Compare a complete implementation after attempting the lab

```python
class TrailCounter:
    distance_unit = "km"

    def __init__(self, trail_name):
        cleaned_name = trail_name.strip()
        if not cleaned_name:
            raise ValueError("trail_name cannot be blank")
        self.trail_name = cleaned_name
```

PYTHON (CONTINUED)

```
        self.distance_km = 0.0
        self.checkpoints = []

    def add_distance(self, amount):
        if amount < 0:
            raise ValueError("amount cannot be negative")
        self.distance_km += amount
        return self.distance_km

    def reach(self, checkpoint):
        cleaned = checkpoint.strip()
        if not cleaned:
            raise ValueError("checkpoint cannot be blank")
        self.checkpoints.append(cleaned)
        return len(self.checkpoints)

    def summary(self):
        return (
            f"{self.trail_name}: {self.distance_km} {self.distance_unit}, "
            f"{len(self.checkpoints)} checkpoints"
        )


ridge = TrailCounter("Ridge Run")
marsh = TrailCounter("Marsh Walk")

assert ridge.add_distance(2.5) == 2.5
assert ridge.add_distance(1.0) == 3.5
assert ridge.reach("Stone Arch") == 1
assert ridge.reach("North Lookout") == 2
assert ridge.summary() == "Ridge Run: 3.5 km, 2 checkpoints"
assert marsh.summary() == "Marsh Walk: 0.0 km, 0 checkpoints"
assert ridge.checkpoints is not marsh.checkpoints
```

The evolving fields belong to each instance. The immutable unit label is shared intentionally. `ridge` becomes `self` for its bound call. The class gives the state one named owner and makes valid operations discoverable at the call site.

Checkpoint: decide whether the class earns its place

Choose state ownership deliberately

Choose the design supported by the behavior and ownership described.

1. Every expedition must own a different mutable checkpoint list. Where should the empty list normally be assigned?

- **A.** As a global variable
- **B.** In a comment above the class
- **C.** To `self.checkpoints` inside `__init__`
- **D.** To `checkpoints` in the class body

2. `distance_unit = "km"` **is on the class. What happens after** `dawn.distance_unit = "miles"`**?**

- **A.** Every instance changes to miles.
- **B.** Python deletes the class.
- **C.** The assignment is forbidden.
- **D.** `dawn` gets a shadowing instance attribute while other instances still find `km` on the class.

3. Which situation gives the strongest reason to introduce a class?

- **A.** Several operations must maintain the valid evolving state of one expedition.
- **B.** One expression adds two numbers.
- **C.** A function converts one string to uppercase.
- **D.** A tuple holds two fixed coordinates and needs no named behavior.

Write your choices before checking the answer key at the end of this book.

8. Key points for your first useful class

- Begin with the simplest working design. Introduce a class when named state ownership and related operations improve the program.
- A class statement creates a class object. Calling the class returns an instance after initialization.
- `__init__` initializes an instance and returns `None`; it is not the method to return `self` from.
- Attributes assigned to `self` belong directly to that instance. Two class calls can therefore produce independent state.
- Functions defined on a class become bound methods when accessed through an instance. The bound instance is supplied as `self`.
- Class attributes are useful for intentionally shared settings and methods. Mutable per-instance state belongs in initialization.
- An instance attribute can shadow a class attribute. `vars` helps you inspect where ordinary state is stored.
- A useful class makes responsibilities and valid operations easier to see. It does not earn its place merely by wrapping a dictionary.

References

- Python tutorial: A first look at classes
- Python tutorial: Class and instance variables
- Python data model: Custom classes
- Python data model: Instance methods
- University of Helsinki: Defining classes

Keep Objects in Valid States

Online lesson and runnable notebook

Turn domain rules into constructor and method checks, make failed updates atomic, and choose plain attributes, properties, methods, and alternate constructors deliberately.

- **Level:** Python Foundations
- **Estimated time:** 4–6 hours
- **You will learn:** Define an object’s invariants, validate construction and every mutation path, keep failed operations atomic, and choose among a plain attribute, property, method, and classmethod constructor.
- **Practice in:** Google Colab, JupyterLab, or a local Python interpreter

A field robot carries a battery rated for 100 energy units. The first class is easy to write:

```python
class Battery:
    def __init__(self, charge):
        self.charge = charge
```

It is also easy to put into impossible states:

```python
battery = Battery(80)
battery.charge = -20
print(battery.charge)
```

Python accepts `-20` because the class has not expressed any rule against it. This lesson turns rules such as “charge stays between zero and capacity” into a public interface that remains trustworthy after successful and failed calls.

Answer these questions as you work:

- What must always be true about a valid object?
- Which operations can change that truth?
- How can a method reject a change without committing half of it?
- When should callers read a property or call a method?
- How can one class support a second input format without duplicating setup?

1. Write the rules before hiding the fields

An **invariant** is a condition that must be true after construction and after every public operation finishes. For this battery:

1. capacity is greater than zero;
2. charge is at least zero; and
3. charge is no greater than capacity.

Write representative examples before the implementation:

```python
# Valid examples
# Battery(capacity=100, charge=60)
# Battery(capacity=25, charge=0)
# Battery(capacity=25, charge=25)

# Invalid examples
# Battery(capacity=0, charge=0)
# Battery(capacity=100, charge=-1)
# Battery(capacity=100, charge=101)
```

This list prevents an implementation from validating only the pleasant case. The boundary values zero and capacity are valid; values just outside them are not.

Now validate before assigning state:

```python
class Battery:
    def __init__(self, capacity, charge):
        if capacity <= 0:
            raise ValueError("capacity must be positive")
        if not 0 <= charge <= capacity:
            raise ValueError("charge must be between 0 and capacity")

        self._capacity = capacity
        self._charge = charge
```

The underscore in `_charge` communicates “implementation detail; use the public interface.” It does not make the value secret or inaccessible:

```python
battery = Battery(100, 60)
print(battery._charge)
```

Python prints `60`. The underscore is a collaboration convention, not a security boundary. The class must still keep every public operation correct.

Construction should not begin invalid

Try the boundaries deliberately:

```python
valid = [Battery(100, 0), Battery(100, 100)]
print(len(valid))

for capacity, charge in [(0, 0), (100, -1), (100, 101)]:
    try:
        Battery(capacity, charge)
    except ValueError as error:
        print(f"{capacity=}, {charge=}: {error}")
```

The constructor rejects each impossible combination. It validates local inputs before assigning any attributes, which also makes the order easy to reason about.

Checkpoint: state the invariant precisely

Identify valid construction

Choose the answer that preserves every battery rule.

1. For a battery with capacity 20, which initial charge is valid?

- **A.** -1
- **B.** 20
- **C.** 21
- **D.** Only 10 is valid

2. What makes `0 <= charge <= capacity` an invariant rather than only an input hint?

- **A.** It appears in a comment.
- **B.** It applies only while printing.
- **C.** It must remain true after construction and every public operation.
- **D.** It is stored as a class name.

3. What does the leading underscore in `_charge` communicate?

- **A.** Python encrypts the value.
- **B.** The attribute can be read only once.
- **C.** Only subclasses may access it.
- **D.** It is a non-public implementation detail by convention, not enforced privacy.

Write your choices before checking the answer key at the end of this book.

2. Give every state change an intention-revealing operation

If callers assign `_charge` directly, they can bypass the rules. Give common changes names from the battery domain:

PYTHON

```
class Battery:
    def __init__(self, capacity, charge):
        if capacity <= 0:
            raise ValueError("capacity must be positive")
        if not 0 <= charge <= capacity:
            raise ValueError("charge must be between 0 and capacity")
        self._capacity = capacity
        self._charge = charge

    def drain(self, amount):
        if amount < 0:
            raise ValueError("amount cannot be negative")
        if amount > self._charge:
            raise ValueError("not enough charge")
        self._charge -= amount
        return self._charge

    def recharge(self, amount):
        if amount < 0:
            raise ValueError("amount cannot be negative")
        if self._charge + amount > self._capacity:
            raise ValueError("charge would exceed capacity")
        self._charge += amount
        return self._charge
```

The method names describe events, not storage mechanics. `drain(8)` is clearer than `set_charge(get_charge() - 8)`, and the class gets one chance to validate the whole transition.

```python
battery = Battery(100, 60)

assert battery.drain(15) == 45
assert battery.recharge(20) == 65
```

List every path that can affect the invariant:

Path	Precondition	State after success	Failure
construction	positive capacity; charge in range	supplied valid values	`ValueError`; no usable object
`drain(amount)`	non-negative and available	charge decreases	`ValueError`; state unchanged
`recharge(amount)`	non-negative and fits	charge increases	`ValueError`; state unchanged

If you later add `reset`, `transfer_to`, or a property setter, add that path to the audit. An invariant enforced in only one method is not an invariant.

3. Validate the whole operation before committing any of it

Suppose an expedition allocates energy from one battery to another. This implementation mutates too early:

```python
def broken_transfer(source, target, amount):
    source.drain(amount)
    target.recharge(amount)
```

If the source has enough charge but the target lacks capacity, `source.drain` succeeds before `target.recharge` fails:

```python
source = Battery(100, 50)
target = Battery(100, 95)

try:
    broken_transfer(source, target, 10)
except ValueError as error:
    print(error)

print(source._charge, target._charge)
```

The output is `40 95`. Ten units vanished. Each object preserved its local range invariant, but the transfer operation violated the larger rule that energy moves as one transaction.

Put the operation with an owner that can validate both sides before mutation. For now a function is sufficient:

```python
def transfer(source, target, amount):
    """Move energy atomically and return both new charges."""
    if amount < 0:
        raise ValueError("amount cannot be negative")
    if amount > source._charge:
        raise ValueError("source lacks charge")
```

PYTHON (CONTINUED)

```
    if target._charge + amount > target._capacity:
        raise ValueError("target lacks capacity")

    source._charge -= amount
    target._charge += amount
    return source._charge, target._charge
```

Check first, commit second:

PYTHON

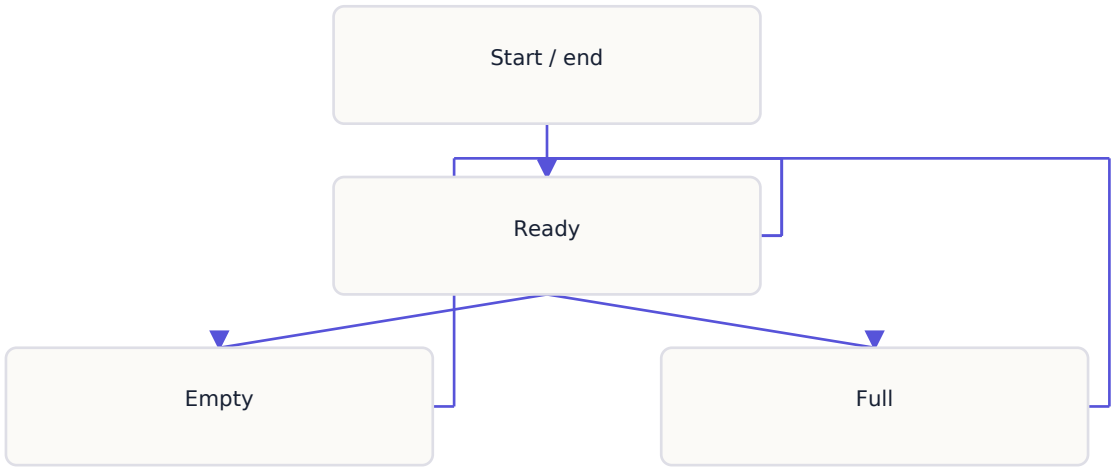
```
source = Battery(100, 50)
target = Battery(100, 95)

before = (source._charge, target._charge)
try:
    transfer(source, target, 10)
except ValueError:
    pass

after = (source._charge, target._charge)
assert after == before
```

This is **atomic at the model level**: the operation either makes the complete valid change or makes no change. It is not a database transaction and does not address threads or processes. It is still a powerful design rule for ordinary Python objects.

Validation is a gate before mutation. A rejected request returns to the same state.



Ready → Ready: valid drain or recharge

Ready → Ready: invalid request rejected

Ready → Empty: drain remaining charge

Empty → Ready: recharge

Ready → Full: recharge to capacity

Full → Ready: drain

Checkpoint: keep failure atomic

Protect state during failed operations

Choose the repair or result that preserves the complete contract.

1. A transfer can fail because the target is full. What should happen before either battery changes?

- **A.** Validate the amount, source charge, and target capacity.
- **B.** Drain the source and hope the target accepts it.
- **C.** Set both charges to zero.
- **D.** Print a warning and continue.

2. A valid source has 50, a nearly full target has 95/100, and a 10-unit atomic transfer is rejected. What state should remain?

- **A.** 40 and 95
- **B.** 50 and 95
- **C.** 50 and 100
- **D.** 40 and 100

3. You add a public `reset()` method. What must the invariant audit do next?

- **A.** Ignore it because construction already validates.
- **B.** Delete the invariant.
- **C.** Include `reset()` as another path that must finish with valid state.
- **D.** Move every field to the class.

Write your choices before checking the answer key at the end of this book.

4. Use a property when attribute syntax tells the truth

Callers need to read charge and capacity without depending on underscore names. Read-only properties expose those values:

PYTHON

```
class Battery:
    def __init__(self, capacity, charge):
        if capacity <= 0:
            raise ValueError("capacity must be positive")
        if not 0 <= charge <= capacity:
            raise ValueError("charge must be between 0 and capacity")
        self._capacity = capacity
        self._charge = charge

    @property
    def capacity(self):
        return self._capacity

    @property
    def charge(self):
        return self._charge

    @property
```

PYTHON (CONTINUED)

```
    def percentage(self):
        return self._charge / self._capacity * 100

    @property
    def is_low(self):
        return self.percentage < 20
```

The caller uses attribute syntax:

PYTHON

```
battery = Battery(80, 12)

print(battery.charge)
print(battery.percentage)
print(battery.is_low)
```

`percentage` and `is_low` are computed from source state. Storing them as separate fields would create synchronization work: every charge change would also need to update two derived fields.

There is no `@charge.setter`, so ordinary assignment is rejected:

PYTHON

```
try:
    battery.charge = 500
except AttributeError as error:
    print(type(error).__name__)
```

This is useful because the domain operations are `drain` and `recharge`, not arbitrary replacement. A property setter would hide which kind of change the caller intends.

Do not manufacture getters and setters

This interface adds ceremony without a rule:

PYTHON

```
class Label:
    def __init__(self, text):
        self._text = text

    def get_text(self):
        return self._text

    def set_text(self, text):
        self._text = text
```

If any text is valid and no behavior accompanies assignment, start with `self.text = text`. Python can later migrate `text` to a property while keeping the caller's `label.text` syntax.

A validating property setter can be appropriate when replacing the value is the honest operation:

PYTHON

```
class TemperatureSetting:
    def __init__(self, celsius):
        self.celsius = celsius

    @property
    def celsius(self):
        return self._celsius

    @celsius.setter
```

PYTHON (CONTINUED)

```
    def celsius(self, value):
        if not -50 <= value <= 50:
            raise ValueError("celsius must be between -50 and 50")
        self._celsius = value
```

The constructor deliberately assigns through the property, reusing one validation path:

PYTHON

```
setting = TemperatureSetting(18)
setting.celsius = 21
assert setting.celsius == 21
```

Choose by meaning:

Interface	Use when
plain attribute	direct reading/replacement is valid and needs no rule
read-only property	callers need attribute-like derived or protected data
property setter	replacing one value through attribute syntax is the honest operation
method	the action has domain meaning, arguments, multiple effects, or important failure

5. Offer alternate construction without duplicating initialization

Suppose configuration stores a battery as `"60/100"`: charge first, capacity second. Do not put parsing branches into every caller. A classmethod can provide a named alternate constructor:

PYTHON

```
class Battery:
    def __init__(self, capacity, charge=0):
        if capacity <= 0:
            raise ValueError("capacity must be positive")
        if not 0 <= charge <= capacity:
            raise ValueError("charge must be between 0 and capacity")
        self._capacity = capacity
        self._charge = charge

    @classmethod
    def from_text(cls, specification):
        """Build a battery from `<charge>/<capacity>` text."""
        charge_text, separator, capacity_text = specification.partition("/")
        if not separator:
            raise ValueError("specification must be <charge>/<capacity>")
        try:
            charge = int(charge_text)
            capacity = int(capacity_text)
        except ValueError as error:
            raise ValueError("charge and capacity must be integers") from error
        return cls(capacity=capacity, charge=charge)

    @property
    def charge(self):
        return self._charge
```

PYTHON (CONTINUED)

```
    @property
    def capacity(self):
        return self._capacity
```

PYTHON

```
battery = Battery.from_text("60/100")
assert battery.charge == 60
assert battery.capacity == 100
```

When `Battery.from_text(...)` is called, Python binds the receiving class to `cls`, just as an instance method binds an instance to `self`. Returning `cls(...)` reuses the primary constructor and supports subclasses that inherit the alternate constructor.

A module-level `parse_battery(text)` function could also be clear. Use a classmethod when the operation's main promise is "construct this class through a named alternate format." Use a function when parsing has a broader responsibility or produces several possible types.

`staticmethod` exists for a function stored in a class namespace without receiving `self` or `cls`. Do not use it merely because a helper is vaguely related. A module-level function is often easier to find and reuse.

6. Make failures and return values part of the interface

Validation is not complete until callers know how failure appears. Use `TypeError` when an operation receives the wrong kind of value and `ValueError` when the kind is acceptable but the value violates the domain range. Do not write a broad `except Exception` inside the object and silently continue with old or partial state.

For example, this helper separates an integer contract from its range:

PYTHON

```
def require_non_negative_integer(value, label):
    if not isinstance(value, int) or isinstance(value, bool):
        raise TypeError(f"{label} must be an integer")
    if value < 0:
        raise ValueError(f"{label} cannot be negative")
```

`bool` is technically a subclass of `int` in Python, so the explicit boolean check rejects `True` as one unit of energy when the domain does not want that surprise. Do not add such checks everywhere by reflex; add them when the public boundary truly promises an integer count.

Return values also belong to the contract. A command-like method can return `None`, a new state value, or an event record. Pick one meaning and use it consistently. These two methods communicate different questions:

PYTHON

```
class Meter:
    def __init__(self, value=0):
        self._value = value

    def add(self, amount):
        """Change the meter and return the new value."""
        self._value += amount
        return self._value

    def is_above(self, boundary):
        """Answer a question without changing the meter."""
        return self._value > boundary
```

Callers should not have to guess whether `is_above` mutates or whether `add` returns the old value. Names, docstrings, examples, and assertions make that contract observable.

When a public operation fails, verify three things separately:

1. the exception type and message identify the violated rule;
2. protected state matches its before snapshot; and
3. a later valid operation still succeeds.

The third check catches objects that leave behind a hidden “busy” flag or other partial transition even when visible fields appear unchanged.

7. Build an energy cell with one trustworthy interface

Implement `EnergyCell`:

```python
class EnergyCell:
    def __init__(self, capacity, charge=0):
        """Create a cell with positive capacity and charge in range."""
        ...

    @classmethod
    def from_percentage(cls, capacity, percentage):
        """Create a cell whose charge is the integer percentage of capacity."""
        ...

    @property
    def capacity(self):
        ...

    @property
    def charge(self):
        ...

    @property
    def percentage(self):
        ...

    def drain(self, amount):
        """Remove available charge and return the remaining charge."""
        ...

    def transfer_to(self, other, amount):
        """Move charge atomically and return both remaining charges."""
        ...
```

Rules:

- capacity is a positive integer;
- charge remains between zero and capacity;
- percentage must be between zero and 100;
- `from_percentage` uses `int(capacity * percentage / 100)`;
- transfer validates both cells before either changes;
- public properties are read-only; and
- every rejected operation leaves all involved state unchanged.

Use these checks:

```python
source = EnergyCell.from_percentage(80, 75)
target = EnergyCell(50, 10)

assert source.capacity == 80
assert source.charge == 60
assert source.percentage == 75.0
assert source.drain(5) == 55
assert source.transfer_to(target, 20) == (35, 30)
assert (source.charge, target.charge) == (35, 30)

before = (source.charge, target.charge)
try:
    source.transfer_to(target, 30)
except ValueError as error:
    assert str(error) == "target lacks capacity"
else:
    raise AssertionError("overfilling transfer should fail")
assert (source.charge, target.charge) == before

try:
    source.charge = 70
except AttributeError:
    pass
else:
    raise AssertionError("charge should be read-only")
```

Hint 1

Store `_capacity` and `_charge` only after validating both. Properties simply return those source fields; `percentage` computes rather than stores.

Hint 2

Let `from_percentage` validate the percentage, calculate charge, and call `cls(capacity, charge)`. Do not duplicate the capacity/charge invariant there.

Hint 3

For `transfer_to`, validate amount, source charge, and `other.charge + amount <= other.capacity` before subtracting or adding. Commit the two assignments only after every check passes.

Compare a complete energy-cell implementation

```python
class EnergyCell:
    def __init__(self, capacity, charge=0):
        if not isinstance(capacity, int) or isinstance(capacity, bool):
            raise TypeError("capacity must be an integer")
        if capacity <= 0:
            raise ValueError("capacity must be positive")
        if not 0 <= charge <= capacity:
            raise ValueError("charge must be between 0 and capacity")
        self._capacity = capacity
        self._charge = charge

    @classmethod
    def from_percentage(cls, capacity, percentage):
        if not 0 <= percentage <= 100:
            raise ValueError("percentage must be between 0 and 100")
        charge = int(capacity * percentage / 100)
        return cls(capacity, charge)
```

PYTHON (CONTINUED)

```
    @property
    def capacity(self):
        return self._capacity

    @property
    def charge(self):
        return self._charge

    @property
    def percentage(self):
        return self._charge / self._capacity * 100

    def drain(self, amount):
        if amount < 0:
            raise ValueError("amount cannot be negative")
        if amount > self._charge:
            raise ValueError("not enough charge")
        self._charge -= amount
        return self._charge

    def transfer_to(self, other, amount):
        if amount < 0:
            raise ValueError("amount cannot be negative")
        if amount > self._charge:
            raise ValueError("source lacks charge")
        if other.charge + amount > other.capacity:
            raise ValueError("target lacks capacity")
        self._charge -= amount
        other._charge += amount
        return self._charge, other._charge


source = EnergyCell.from_percentage(80, 75)
target = EnergyCell(50, 10)
assert source.drain(5) == 55
assert source.transfer_to(target, 20) == (35, 30)
```

The transfer reaches into another instance of the same class to commit the second field only after validation. A larger system could give transfer to a separate service that owns both participants. The important property is the same: one operation validates the complete transition before mutation.

Checkpoint: choose the public interface

Match behavior to an interface

Choose the interface that communicates the stated operation most clearly.

1. `percentage` **is derived from current charge and capacity. What is the clearest default interface?**

- **A.** A mutable class list
- **B.** A method that also drains charge
- **C.** A global updated by every caller
- **D.** A read-only computed property

2. Why is `battery.drain(5)` **usually clearer than assigning** `battery.charge = battery.charge - 5`**?**

- **A.** The method names the domain action and owns its validation.
- **B.** Properties cannot return numbers.

- **C.** Assignments never change objects.
- **D.** Methods automatically make code concurrent.

3. What should `Battery.from_text("60/100")` normally do after parsing the values?

- **A.** Assign directly to unrelated globals.
- **B.** Return `cls(capacity=100, charge=60)` so primary validation is reused.
- **C.** Return the string unchanged.
- **D.** Call `__init__` without an instance and ignore its result.

Write your choices before checking the answer key at the end of this book.

8. Key points for trustworthy object state

- State an invariant as a rule that remains true after construction and every public operation—not as validation on only one input path.
- Validate all reasons an operation can fail before committing its first mutation when the operation must be atomic.
- Use underscore-prefixed attributes to communicate a non-public convention, not to claim enforced privacy.
- Prefer intention-revealing methods for domain actions such as draining or transferring energy.
- Use a plain attribute when unrestricted reading and assignment are honest. Use properties for attribute-like computed, read-only, or validated access.
- Compute derived state instead of storing a second value that every mutation must synchronize.
- A classmethod can name an alternate input format and return `cls(...)`, reusing the primary constructor.
- `__init__` initializes valid state and returns `None`.
- A failed operation should leave the object—or all participating objects—in a predictable state.

References

- Python built-in `property`
- Python built-in `classmethod`
- Python tutorial: Private variables
- Python data model: Object creation
- CS50P notes: Properties and class methods

Use Dataclasses for Value-Like Objects

Online lesson and runnable notebook

Replace repetitive field-based methods with dataclasses while controlling validation, defaults, representation, equality, frozen updates, and mutable boundaries.

- **Level:** Python Foundations
- **Estimated time:** 4–5 hours
- **You will learn:** Use a dataclass when field-based value semantics fit, inspect generated methods, create independent defaults, validate with `__post_init__`, and explain the limits of `frozen=True`.
- **Practice in:** Google Colab, JupyterLab, or a local Python interpreter

A map waypoint has a label, row, and column. Two waypoints with the same three values should compare equal, and a printed waypoint should reveal those values. You can implement all of that in a regular class. The question is whether the repetition expresses a unique design or merely repeats a common field-based pattern.

In this lesson you will decide:

- which methods a dataclass should generate;
- which rules still belong to you;
- how equality follows field declarations;
- why mutable defaults need factories;
- when representation should include or exclude a field; and
- what “frozen” protects—and what it does not.

1. Write the repetitive class once

Begin without a decorator:

PYTHON

```
class Waypoint:
    def __init__(self, label, row, column):
        self.label = label
        self.row = row
        self.column = column

    def __repr__(self):
        return (
            f"Waypoint(label={self.label!r}, "
            f"row={self.row!r}, column={self.column!r})"
        )

    def __eq__(self, other):
        if not isinstance(other, Waypoint):
            return NotImplemented
        return (
            self.label,
            self.row,
```

PYTHON (CONTINUED)

```
            self.column,
        ) == (
            other.label,
            other.row,
            other.column,
        )
```

The methods are useful:

PYTHON

```
first = Waypoint("River Gate", 2, 4)
same = Waypoint("River Gate", 2, 4)
other = Waypoint("Hill Gate", 2, 4)

print(first)
print(first == same)
print(first == other)
```

TEXT

```
Waypoint(label='River Gate', row=2, column=4)
True
False
```

`__repr__` supplies a developer-oriented representation used by `repr`, the interactive prompt, and containers. `!r` asks each field for its representation so strings retain quotes. `__eq__` defines value equality for `==`.

None of this code validates what makes a coordinate meaningful. Most of it copies field names into predictable positions. That is the repetitive part a dataclass can generate.

2. Let the field declarations drive generated methods

Apply `@dataclass`:

PYTHON

```
from dataclasses import dataclass


@dataclass
class Waypoint:
    label: str
    row: int
    column: int
```

The annotated lines declare dataclass fields in order. With its default settings, the decorator generates an initializer, representation, and equality method similar to the manual versions.

PYTHON

```
import inspect

print(inspect.signature(Waypoint))

first = Waypoint("River Gate", 2, 4)
same = Waypoint("River Gate", 2, 4)

print(first)
print(first == same)
```

The signature exposes `label`, `row`, and `column` in declaration order. The representation identifies the class and its field values. Equality is `True` because all compared fields match.

The annotations are metadata. They improve documentation, editor support, and later static checking, but a dataclass does not enforce them at runtime:

PYTHON
```
surprising = Waypoint(label=99, row="north", column=None)
print(surprising)
```

Python creates the instance unless you add runtime validation. Unit 14 explores static type checking; this lesson will add domain validation with `__post_init__`.

Generated equality has an exact-type boundary

Dataclass equality requires both objects to have the identical concrete class:

PYTHON
```
@dataclass
class Destination:
    label: str
    row: int
    column: int


waypoint = Waypoint("River Gate", 2, 4)
destination = Destination("River Gate", 2, 4)

print(waypoint == destination)
```

The result is `False`. Matching field names and values do not erase the different meanings of `Waypoint` and `Destination`.

Within the same class, every field with `compare=True` participates in declaration order. Adding a field can therefore change equality. That is an API decision, not a harmless formatting change.

Checkpoint: inspect generated behavior

Identify what the decorator generated

Choose the result defined by the dataclass field declarations.

1. Which useful methods does a default `@dataclass` generate for the `Waypoint` fields?

- **A.** Only `__str__`
- **B.** A file reader and writer
- **C.** An initializer, representation, and equality method
- **D.** Every domain validation rule

2. What happens by default if `Waypoint(row="north", ...)` receives a string despite `row: int`?

- **A.** The string is always converted to an integer.
- **B.** Python deletes the object.
- **C.** The decorator raises before `__init__`.
- **D.** Runtime construction can succeed unless validation is added.

3. A `Waypoint` and `Destination` declare identical fields and values. What does generated equality return?

- **A.** `False`, because generated equality requires the same concrete class
- **B.** `True`, because field names are enough
- **C.** Their row difference
- **D.** A new combined object

Write your choices before checking the answer key at the end of this book.

3. Put defaults after required fields

Fields without defaults must come before fields with defaults:

```python
@dataclass
class Landmark:
    name: str
    row: int
    column: int
    discovered: bool = False
```

The constructor now allows both forms:

```python
hidden = Landmark("Moon Arch", 5, 8)
known = Landmark("Old Bridge", 2, 1, discovered=True)

print(hidden)
print(known)
```

Placing `name: str` after `discovered: bool = False` would make a required parameter follow a defaulted one. The decorator reports a `TypeError`, matching the parameter-order rule for ordinary functions.

4. Give every instance its own mutable default

A route plan needs a list of stops. This definition is rejected by modern Python dataclasses:

```python
from dataclasses import dataclass


try:
    @dataclass
    class BrokenRoute:
        name: str
        stops: list = []
except ValueError as error:
    print(type(error).__name__)
    print(error)
```

The direct list default would be created once while the class is defined, not once per instance. It is the dataclass version of the shared class-list bug from Lesson 1.

Use `field(default_factory=list)`:

```python
from dataclasses import dataclass, field


@dataclass
class RoutePlan:
    name: str
    stops: list[str] = field(default_factory=list)
```

The factory is the callable `list`, not the result `list()`. The generated initializer calls it when no value is supplied:

```python
north = RoutePlan("North Route")
south = RoutePlan("South Route")

north.stops.append("River Gate")

print(north)
print(south)
print(north.stops is south.stops)
```

The second list remains empty and the identity check is `False`.

A factory can also build a non-empty default when that default must be fresh:

```python
def starting_notes():
    return ["Check supplies"]


@dataclass
class JourneyLog:
    title: str
    notes: list[str] = field(default_factory=starting_notes)
```

Keep the factory deterministic and side-effect free. A factory that reads a file or asks for input would hide important work inside ordinary construction.

Checkpoint: control field defaults

Keep dataclass instances independent

Choose the field declaration or effect that matches the default contract.

1. Which declaration gives each `RoutePlan` its own empty list?

- **A.** `stops: list = []`
- **B.** `stops: list = field(default_factory=list)`
- **C.** `stops = RoutePlan.stops`
- **D.** `stops: list = list`

2. After two default `RoutePlan` instances are created, what should `north.stops is south.stops` be?

- **A.** `True`, because both are lists
- **B.** The route name

- **C.** `False`, because the factory created two lists
- **D.** A field-definition error

3. Why must required dataclass fields normally appear before fields with defaults?

- **A.** Representations sort alphabetically.
- **B.** Equality ignores the first field.
- **C.** Lists must be last in every class.
- **D.** The generated initializer follows Python's parameter rule against required arguments after defaults.

Write your choices before checking the answer key at the end of this book.

5. Validate generated initialization with `__post_init__`

The generated `__init__` assigns fields, then calls `__post_init__` when that method exists. Use it for domain relationships among fields:

PYTHON

```
@dataclass
class RouteSegment:
    start: str
    end: str
    distance_km: float

    def __post_init__(self):
        self.start = self.start.strip()
        self.end = self.end.strip()
        if not self.start or not self.end:
            raise ValueError("start and end cannot be blank")
        if self.start == self.end:
            raise ValueError("start and end must differ")
        if self.distance_km <= 0:
            raise ValueError("distance_km must be positive")
```

PYTHON

```
segment = RouteSegment(" River Gate ", "Hill Camp", 4.5)
assert segment.start == "River Gate"

try:
    RouteSegment("Camp", "Camp", 2)
except ValueError as error:
    print(error)
```

Dataclasses remove method boilerplate; they do not remove invariant design. `__post_init__` still needs precise exceptions and boundary examples.

If normalization changes the meaning supplied by the caller, document it. A different design may reject surrounding spaces instead. The class should not silently invent policy just because a hook is available.

6. Shape diagnostic and user-facing representations

Field representations are helpful until one field is noisy or sensitive:

```python
@dataclass
class AccessBadge:
    owner: str
    zone: str
    access_code: str = field(repr=False)
```

```python
badge = AccessBadge("Ari", "Archive", "MOON-742")
print(repr(badge))
```

The generated representation omits `access_code`. This avoids casual display; it is not encryption. The value still exists in memory and can be accessed by code.

Use `compare=False` only when a field genuinely does not contribute to value meaning:

```python
@dataclass
class Reading:
    sensor: str
    value: float
    received_note: str = field(default="", compare=False)
```

Two readings with the same sensor and value compare equal even if their notes differ. That may be correct for a transient import note; it would be wrong if the timestamp or unit were essential to the reading's meaning.

`__repr__` serves developers and diagnosis. Add `__str__` when a distinct user-facing display is useful:

```python
@dataclass
class MapTile:
    terrain: str
    row: int
    column: int

    def __str__(self):
        return f"{self.terrain} at ({self.row}, {self.column})"
```

```python
tile = MapTile("forest", 3, 7)
print(repr(tile))
print(str(tile))
```

The diagnostic representation retains class and field names. The user-facing text reads as a sentence. Do not make `__repr__` vague solely to look pretty; debugging needs evidence.

7. Freeze field rebinding without promising deep immutability

Coordinates are useful immutable values:

```python
@dataclass(frozen=True)
class Position:
    row: int
    column: int
```

Ordinary field assignment raises `FrozenInstanceError`:

```python
from dataclasses import FrozenInstanceError

position = Position(2, 4)

try:
    position.row = 3
except FrozenInstanceError as error:
    print(type(error).__name__)
```

Create a changed value with `dataclasses.replace`:

```python
from dataclasses import replace

moved = replace(position, row=position.row + 1)

assert moved == Position(3, 4)
assert position == Position(2, 4)
```

The old value remains unchanged. This style works well for coordinates, configuration values, and result records.

“Frozen” is shallow. A frozen field can still refer to a mutable object:

```python
@dataclass(frozen=True)
class FrozenRoute:
    name: str
    stops: list[str] = field(default_factory=list)


route = FrozenRoute("North")
route.stops.append("River Gate")
print(route)
```

The field cannot be rebound normally, but the list object itself remains mutable. Use an immutable field value such as a tuple when the whole value must remain stable:

```python
@dataclass(frozen=True)
class Route:
    name: str
    stops: tuple[str, ...] = ()
```

With equality enabled, a mutable dataclass is normally unhashable because changing compared fields after dictionary/set placement would break lookup. A frozen dataclass can receive a generated hash when all compared fields are hashable. A frozen dataclass containing a list is still unhashable. Connect this to Unit 6: hashability depends on the complete value graph, not the decorator’s name alone.

8. Choose between a record and an entity

Use this decision table as a starting point:

Question	Likely choice
Is it only a short local bundle of values?	tuple or dictionary

Question	Likely choice
Do named fields, useful `repr`, and field equality express its meaning?	dataclass
Should updates produce a new value?	frozen dataclass plus `replace` or a method returning a new value
Does one identity evolve through several guarded operations?	regular stateful class, possibly a dataclass only if generated field behavior remains honest
Would generated equality accidentally claim two entities are interchangeable?	regular class with identity semantics or deliberate equality

A player in a game may have identity and evolving state; two players with the same name and row are not necessarily interchangeable. A `Position(2, 4)` is a value; any other `Position(2, 4)` can mean the same coordinate. That difference matters more than the number of fields.

Do not generate ordering until the domain defines it

`@dataclass(order=True)` can generate `<`, `<=`, `>`, and `>=` from the same field tuple used for equality. That convenience is safe only when declaration order matches the domain's one natural ordering.

PYTHON
```
@dataclass(order=True, frozen=True)
class RaceTime:
    minutes: int
    seconds: int

    def __post_init__(self):
        if self.minutes < 0 or not 0 <= self.seconds < 60:
            raise ValueError("invalid race time")
```

PYTHON
```
assert RaceTime(4, 15) < RaceTime(5, 0)
```

Minutes followed by seconds forms the intended comparison key. Now consider a map tile declared as `terrain, row, column, danger`. Field-tuple ordering would put terrain spelling before location or danger, but the domain has no obvious reason to call `forest` less than `river`. Leave ordering disabled and use an explicit key for the particular question:

PYTHON
```
tiles = [
    MapTile("forest", 2, 1),
    MapTile("river", 1, 4),
]

by_location = sorted(tiles, key=lambda tile: (tile.row, tile.column))
```

An explicit key names the current ordering purpose and can change without changing every comparison of the class.

Keep class settings out of generated fields

Only annotated attributes are normally treated as dataclass fields. An unannotated class setting can document a shared fixed rule:

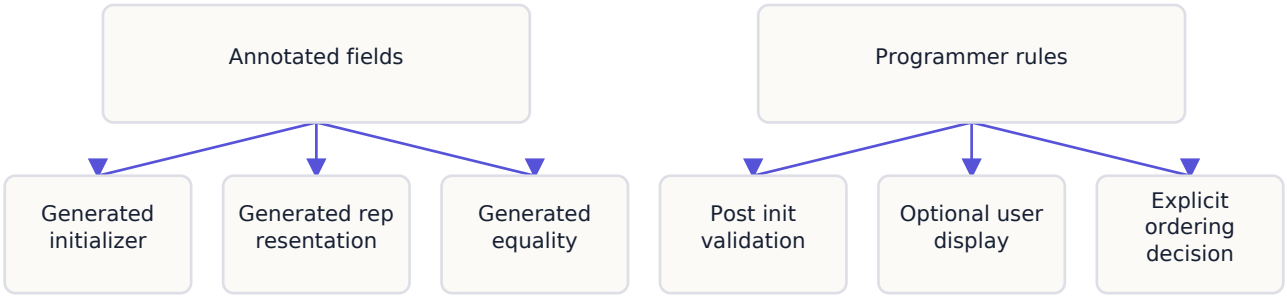
```python
@dataclass(frozen=True)
class CappedTile:
    MAX_DANGER = 5

    terrain: str
    danger: int

    def __post_init__(self):
        if not 0 <= self.danger <= self.MAX_DANGER:
            raise ValueError("danger is outside the supported range")
```

`MAX_DANGER` does not appear in the initializer, representation, or equality tuple. Later typing lessons introduce `ClassVar`, which marks that intent for type checkers. For now, inspect the generated signature whenever a class mixes field declarations and shared settings.

The decorator derives mechanical behavior from field declarations. Domain validation, user display, and ordering policy remain explicit design choices.



9. Model a safe tile and a changing route plan

Create two types with different semantics:

```python
@dataclass(frozen=True)
class MapTile:
    terrain: str
    row: int
    column: int
    danger: int = 0

    def __post_init__(self):
        """Normalize terrain and require non-negative coordinates and danger 0–5."""
        ...

    def __str__(self):
        """Return `<terrain> at (<row>, <column>)`."""
        ...


@dataclass
class RoutePlan:
    name: str
    tiles: list[MapTile] = field(default_factory=list)

    def add(self, tile):
        """Append a MapTile once and return the number of tiles."""
        ...
```

PYTHON (CONTINUED)

```
    @property
    def total_danger(self):
        """Return the sum of danger values without storing duplicate state."""
        ...
```

Checks:

PYTHON

```
forest = MapTile(" Forest ", 1, 2, danger=2)
same_forest = MapTile("forest", 1, 2, danger=2)
bridge = MapTile("bridge", 1, 3, danger=1)

assert forest == same_forest
assert str(forest) == "forest at (1, 2)"

north = RoutePlan("North")
south = RoutePlan("South")
assert north.tiles is not south.tiles
assert north.add(forest) == 1
assert north.add(bridge) == 2
assert north.add(forest) == 2
assert north.total_danger == 3
assert south.tiles == []

safer = replace(forest, danger=0)
assert safer == MapTile("forest", 1, 2, danger=0)
assert forest.danger == 2

for bad_values in [
    ("", 0, 0, 0),
    ("forest", -1, 0, 0),
    ("forest", 0, 0, 6),
]:
    try:
        MapTile(*bad_values)
    except ValueError:
        pass
    else:
        raise AssertionError(f"should reject {bad_values}")
```

Hint 1

Because `MapTile` is frozen, normalization in `__post_init__` must use `object.__setattr__(self, "terrain", cleaned)` after validation. This is a controlled initialization technique, not permission for later mutation.

Hint 2

Validate terrain after stripping, require both coordinates to be non-negative, and use `0 <= danger <= 5`. `RoutePlan.add` can use membership, which invokes the tile's generated equality.

Hint 3

Create each route's list with `default_factory=list`. Compute `total_danger` with `sum(tile.danger for tile in self.tiles)` rather than storing a second total field.

Compare a complete value-and-entity solution

PYTHON

```
from dataclasses import dataclass, field, replace
```

PYTHON (CONTINUED)

```
@dataclass(frozen=True)
class MapTile:
    terrain: str
    row: int
    column: int
    danger: int = 0

    def __post_init__(self):
        cleaned = self.terrain.strip().casefold()
        if not cleaned:
            raise ValueError("terrain cannot be blank")
        if self.row < 0 or self.column < 0:
            raise ValueError("coordinates cannot be negative")
        if not 0 <= self.danger <= 5:
            raise ValueError("danger must be between 0 and 5")
        object.__setattr__(self, "terrain", cleaned)

    def __str__(self):
        return f"{self.terrain} at ({self.row}, {self.column})"


@dataclass
class RoutePlan:
    name: str
    tiles: list[MapTile] = field(default_factory=list)

    def add(self, tile):
        if not isinstance(tile, MapTile):
            raise TypeError("tile must be a MapTile")
        if tile not in self.tiles:
            self.tiles.append(tile)
        return len(self.tiles)

    @property
    def total_danger(self):
        return sum(tile.danger for tile in self.tiles)


forest = MapTile(" Forest ", 1, 2, danger=2)
bridge = MapTile("bridge", 1, 3, danger=1)
north = RoutePlan("North")
assert north.add(forest) == 1
assert north.add(bridge) == 2
assert north.total_danger == 3
assert replace(forest, danger=0).danger == 0
```

`MapTile` behaves as a value: fields define equality and changes produce another tile. `RoutePlan` owns a changing sequence and a meaningful `add` operation. Both use dataclass support, but for different reasons.

Checkpoint: choose value semantics deliberately

Decide what value-like means

Choose the answer that respects equality, mutability, and representation.

1. A frozen dataclass has a `stops: list[str]` field. What can still happen?

- **A.** The list itself can still be mutated, because freezing is shallow.
- **B.** Python converts the list to a tuple automatically.
- **C.** Every instance shares the list.
- **D.** The dataclass loses its fields.

2. How should code derive `Position(3, 4)` from frozen `Position(2, 4)` while keeping the original?

- **A.** Assign directly to `position.row`.
- **B.** Use `replace(position, row=3)` to create a new value.
- **C.** Change the class attribute globally.
- **D.** Append the row to the instance.

3. Which concept is the strongest frozen-dataclass candidate?

- **A.** A service that reads keyboard input forever
- **B.** A manager that owns every project task
- **C.** A coordinate whose row and column completely determine its value
- **D.** A unique player whose health changes through many guarded actions

Write your choices before checking the answer key at the end of this book.

10. Key points for dataclass design

- Use a dataclass when declared fields naturally drive initialization, representation, and equality. It is not merely a shorter class syntax.
- Type annotations describe fields but do not enforce runtime types by themselves.
- Generated equality compares fields marked for comparison and requires the same concrete class.
- Use `field(default_factory=...)` for a fresh mutable default per instance.
- `__post_init__` owns validation and normalization after generated assignment.
- `repr=False` and `compare=False` change important behavior; use them only with a stated reason.
- `__repr__` serves diagnosis, while `__str__` may provide a separate user-facing display.
- `frozen=True` prevents normal field rebinding, not mutation inside a referenced list or dictionary.
- Use `replace` to derive changed frozen values. Hashability still requires all compared fields to be hashable.
- A value-like dataclass and a stateful entity answer different design needs, even when both use `@dataclass`.

References

- Python `dataclasses` documentation
- `dataclasses.field`
- `dataclasses.replace`
- Python data model: `object.__repr__`
- Python data model: `object.__eq__`

Choose Composition Before Inheritance

Online lesson and runnable notebook

Split responsibilities among injected collaborators, delegate through shared behavior, test subtype promises, and replace brittle inheritance with clearer composition.

- **Level:** Python Foundations
- **Estimated time:** 5–6 hours
- **You will learn:** Separate responsibilities, inject and replace collaborators, use duck typing and delegation, override behavior safely, and accept inheritance only when every subtype preserves the base contract.
- **Practice in:** Google Colab, JupyterLab, or a local Python interpreter

An `Explorer` object begins by storing a name and energy. Then it chooses routes, formats messages, rolls random weather, saves progress, and sends notifications. The class now changes for unrelated reasons and is hard to exercise without the outside world.

The solution is not automatically a family tree. First ask which responsibilities can collaborate through small, visible behaviors. Composition lets an object receive and use another object. Inheritance says something stronger: every specialized object can stand in for the base object without surprising its callers.

This lesson asks:

- Which object should own each decision?
- What is the difference between containing, depending on, and delegating?
- How can behavior vary without a shared parent class?
- What promise must an inheritance relationship preserve?
- When is a function, dictionary, or dataclass still the simpler choice?

1. Split a class by reasons to change

Here is the beginning of an overloaded design:

PYTHON

```
class Explorer:
    def __init__(self, name, energy=10):
        self.name = name
        self.energy = energy

    def choose_route(self, routes):
        return min(routes, key=lambda route: route["distance"])

    def travel(self, route):
        self.energy -= route["distance"]
        return f"{self.name} travels to {route['name']}"

    def format_status(self):
        return f"{self.name}: {self.energy} energy"
```

This class is not yet enormous, but it already owns three decisions:

1. explorer state and travel rules;
2. route-selection policy; and
3. presentation text.

Counting methods is not enough. A class with twelve cohesive methods can be healthier than a class with three unrelated ones. Ask what would cause the code to change. A new route algorithm should not risk the energy invariant. A new display language should not alter travel.

Move route choice to a collaborator:

```python
class ShortestRoutePlanner:
    def choose(self, routes):
        if not routes:
            raise ValueError("routes cannot be empty")
        return min(routes, key=lambda route: route["distance"])


class Explorer:
    def __init__(self, name, planner, energy=10):
        if energy < 0:
            raise ValueError("energy cannot be negative")
        self.name = name
        self._planner = planner
        self._energy = energy

    @property
    def energy(self):
        return self._energy

    def choose_route(self, routes):
        return self._planner.choose(routes)

    def travel(self, route):
        cost = route["distance"]
        if cost > self._energy:
            raise ValueError("not enough energy")
        self._energy -= cost
        return route["name"]
```

The constructor receives `planner` rather than constructing it. The explorer **depends on** an object with `choose(routes)` behavior. It stores that collaborator and **delegates** route choice to it.

```python
routes = [
    {"name": "River Gate", "distance": 4, "scenic": 3},
    {"name": "Hill Garden", "distance": 6, "scenic": 5},
]

explorer = Explorer("Mina", ShortestRoutePlanner(), energy=10)
route = explorer.choose_route(routes)

assert route["name"] == "River Gate"
assert explorer.travel(route) == "River Gate"
assert explorer.energy == 6
```

This is **dependency injection** in a simple form: the outside caller provides the collaborator. It makes the dependency visible and replaceable.

2. Swap behavior without inventing a parent class

Add a second planner:

```python
class ScenicRoutePlanner:
    def choose(self, routes):
        if not routes:
            raise ValueError("routes cannot be empty")
        return max(routes, key=lambda route: route["scenic"])
```

It does not inherit from `ShortestRoutePlanner`. It simply supports the behavior `Explorer` uses:

```python
scenic_explorer = Explorer("Sol", ScenicRoutePlanner(), energy=10)
route = scenic_explorer.choose_route(routes)

assert route["name"] == "Hill Garden"
```

Python commonly supports **duck typing**: code relies on available behavior rather than requiring an object's class to appear in one hierarchy. If both planners can accept the route collection and return one route under the stated failure contract, the explorer can collaborate with either.

The behavior still needs a contract:

Planner operation	Promise
`choose(routes)`	accept a non-empty iterable of route records
success	return one of the supplied records
empty input	raise `ValueError("routes cannot be empty")`
side effects	do not mutate the supplied records or explorer

A matching method name alone is not enough. This object has a `choose` method but violates the return contract:

```python
class BrokenPlanner:
    def choose(self, routes):
        return "surprise"
```

`Explorer.choose_route` can call it, but later code expecting `route["distance"]` will fail. Behavioral substitution includes inputs, outputs, failures, and side effects.

Distinguish ownership, reference, dependency, and delegation

Object diagrams often draw the same arrow for several relationships. Name the relationship in prose so readers know what lifecycle and mutation to expect:

- **Ownership:** a `Tour` copies and controls its internal stop sequence. The sequence should not change because the caller appends to its original list.
- **Reference:** a `TurnResult` may refer to an immutable position value without controlling a separate lifecycle.
- **Dependency:** an `Explorer` requires planner behavior to complete route choice.

- **Delegation:** `Explorer.choose_route` forwards the selection request to that planner and returns its result.

Composition does not automatically imply exclusive ownership. In the current constructor, the caller creates the planner and the explorer retains a reference to it. If two explorers receive the same stateful planner, they intentionally or accidentally share that collaborator:

```python
class CountingPlanner:
    def __init__(self):
        self.calls = 0

    def choose(self, routes):
        self.calls += 1
        return routes[0]


shared = CountingPlanner()
first_explorer = Explorer("Ari", shared)
second_explorer = Explorer("Bo", shared)

first_explorer.choose_route(routes)
second_explorer.choose_route(routes)
assert shared.calls == 2
```

That can be useful for shared metrics, but the constructor and documentation should make the choice visible. Copying an arbitrary collaborator is rarely a safe fix: a copy may duplicate file handles, caches, or configuration in surprising ways. Decide lifecycle at the design boundary instead.

The caller can also pass a simple fake object while investigating behavior:

```python
class FirstRoute:
    def choose(self, routes):
        return routes[0]


predictable = Explorer("Cy", FirstRoute(), energy=10)
assert predictable.choose_route(routes) is routes[0]
```

No testing framework is required to benefit from a replaceable dependency. The small fake makes the explorer’s delegation observable.

Inject uncertainty instead of hiding it

This version is hard to control:

```python
import random


class HiddenWeatherExplorer:
    def weather(self):
        return random.choice(["clear", "rain"])
```

The object constructs its uncertainty implicitly through the global `random` module. A small collaborator makes examples deterministic:

```python
class SequenceWeather:
    def __init__(self, conditions):
```

PYTHON (CONTINUED)

```
        self._conditions = iter(conditions)

    def next_condition(self):
        return next(self._conditions)


class Expedition:
    def __init__(self, weather):
        self._weather = weather

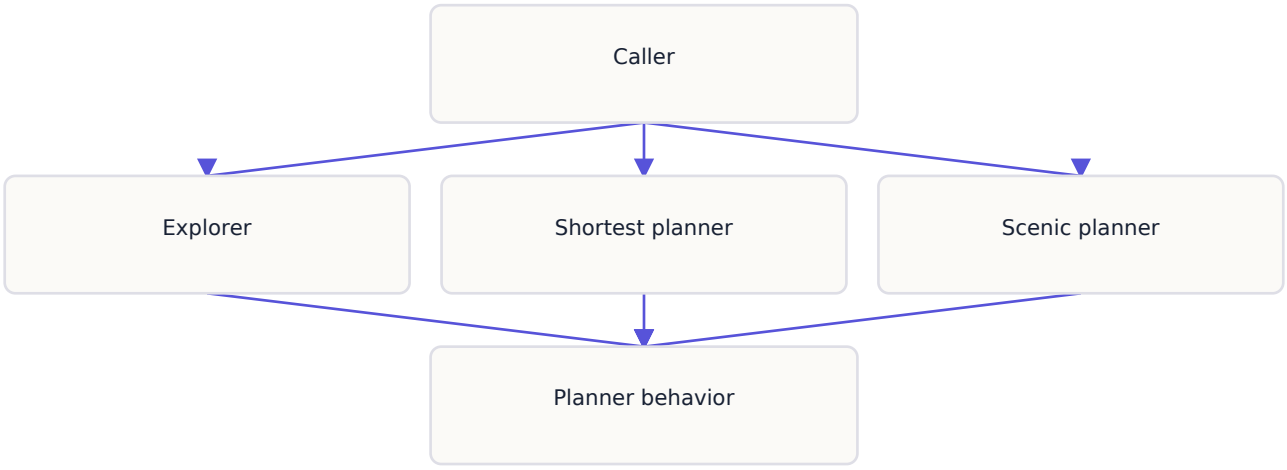
    def begin_day(self):
        return self._weather.next_condition()
```

PYTHON

```
expedition = Expedition(SequenceWeather(["clear", "rain"]))
assert expedition.begin_day() == "clear"
assert expedition.begin_day() == "rain"
```

The expedition owns the workflow; the collaborator owns condition selection. Unit 13 will use this same seam for test doubles. Here, it simply makes the design predictable.

Composition places the policy beside the object that uses it. The caller chooses which compatible policy to supply.



Caller → Explorer: injects

Explorer → Planner behavior: delegates choose

Checkpoint: compose through behavior

Trace injected collaborators

Choose the design statement supported by the examples.

1. Who decides which route is shortest after `Explorer` receives `ShortestRoutePlanner`?

- **A.** The route dictionary mutates itself.
- **B.** The `Explorer` constructor chooses randomly.
- **C.** The display string chooses.

- **D.** The planner's `choose` method; Explorer delegates to it.

2. Why can `ScenicRoutePlanner` work without inheriting from `ShortestRoutePlanner`?

- **A.** It provides the `choose(routes)` behavior and contract the caller uses.
- **B.** Every class inherits from the nearest class name automatically.
- **C.** Dictionaries disable type checks.
- **D.** The explorer never calls the planner.

3. What does supplying a weather collaborator through the constructor make possible?

- **A.** It removes every object from the design.
- **B.** The caller can choose a deterministic condition source without editing the expedition.
- **C.** It makes random output a class attribute.
- **D.** It guarantees internet access.

Write your choices before checking the answer key at the end of this book.

3. Use inheritance only for a reliable subtype

Inheritance defines a new class from an existing class. A subtype receives base behavior and can add or override behavior:

PYTHON
```
class Notification:
    def __init__(self, recipient, message):
        if not recipient.strip():
            raise ValueError("recipient cannot be blank")
        self.recipient = recipient.strip()
        self.message = message

    def render(self):
        return f"To {self.recipient}: {self.message}"


class UrgentNotification(Notification):
    def __init__(self, recipient, message, alarm="ALERT"):
        super().__init__(recipient, message)
        self.alarm = alarm

    def render(self):
        base_text = super().render()
        return f"{self.alarm}! {base_text}"
```

`UrgentNotification(Notification)` declares inheritance. `super().__init__` continues to the base initializer so recipient validation and shared assignments remain in one place. Its `render` override extends the base result.

PYTHON
```
ordinary = Notification("Ari", "Gate opened")
urgent = UrgentNotification("Mina", "Bridge closing")

assert ordinary.render() == "To Ari: Gate opened"
assert urgent.render() == "ALERT! To Mina: Bridge closing"
assert isinstance(urgent, Notification)
```

A caller that accepts a `Notification` and relies only on the public promise can use either:

```python
def display_notification(notification):
    """Return displayable notification text."""
    text = notification.render()
    if not isinstance(text, str) or not text:
        raise ValueError("notification must render non-empty text")
    return text


assert display_notification(ordinary) == "To Ari: Gate opened"
assert display_notification(urgent).startswith("ALERT!")
```

The subtype changes formatting while preserving the operation's input and non-empty-string result. This is a small, honest hierarchy.

`super()` follows the class relationship

`super()` is not "call my parent by name." It creates a proxy that continues method lookup after the current class according to Python's method resolution order. In this shallow example, that reaches `Notification`. Using `super()` supports cooperative behavior better than hard-coding `Notification.__init__(self, ...)`.

This unit stops at single, shallow inheritance. Multiple inheritance and deep method-resolution design need more context and belong in an advanced course.

4. Reject a subtype that breaks the caller's promise

Suppose a base `Storage` promises that `save(text)` accepts any string and returns the stored character count:

```python
class Storage:
    def __init__(self):
        self.items = []

    def save(self, text):
        self.items.append(text)
        return len(text)
```

This subtype is tempting but dishonest:

```python
class NonEmptyStorage(Storage):
    def save(self, text):
        if not text:
            raise ValueError("empty text is forbidden")
        return super().save(text)
```

The base accepts `""`, but the subtype rejects it. Code written for the base contract can fail when handed the subtype:

```python
def archive_values(storage, values):
    return [storage.save(value) for value in values]


assert archive_values(Storage(), ["map", ""]) == [3, 0]

try:
```

PYTHON (CONTINUED)

```
    archive_values(NonEmptyStorage(), ["map", ""])
except ValueError as error:
    print(error)
```

An “is-a storage” sentence is not enough. The subtype strengthened the precondition and broke substitutability. Better options include:

- change the base contract if all storage should reject empty text;
- validate before calling storage, where that application rule belongs; or
- compose a filtering/validation policy with storage rather than pretending it is a subtype.

Use a substitution review:

Contract part	Subtype question
accepted inputs	Does it accept everything the base promises?
successful result	Does it return a result callers can use the same way?
failure behavior	Does it introduce surprising failure for valid base input?
state effects	Does it preserve the documented postconditions?
public attributes/methods	Are inherited promises still meaningful?

Inheritance is a public relationship. A subclass can technically access underscore-prefixed base state, but depending on every internal detail makes the base fragile: a harmless refactor can break subclasses. Design extension points intentionally or prefer composition.

Checkpoint: test the subtype promise

Accept or reject an inheritance relationship

Choose the answer that preserves callers’ behavioral expectations.

1. Why does `UrgentNotification.__init__` call `super().__init__(recipient, message)`?

- **A.** To convert the instance into a function
- **B.** To skip base validation
- **C.** To reuse the base initialization contract before adding urgent state
- **D.** To create a global notification

2. Why is `NonEmptyStorage` not safely substitutable for the stated `Storage` contract?

- **A.** It uses a list.
- **B.** Its class name is too long.
- **C.** It calls `super()`.
- **D.** It rejects an empty string that the base contract accepts.

3. Which override best preserves a base method that promises a non-empty display string?

- **A.** Return another non-empty string with extra urgent context.

- **B.** Return `None` for urgent objects.
- **C.** Require an unrelated file argument.
- **D.** Delete the method after construction.

Write your choices before checking the answer key at the end of this book.

5. Replace a brittle hierarchy with a collaborator

Suppose a game models movement this way:

PYTHON
```
class MovingActor:
    def move(self, position):
        raise NotImplementedError


class WalkingActor(MovingActor):
    def move(self, position):
        return position + 1


class FlyingWalkingActor(WalkingActor):
    def move(self, position):
        return position + 3
```

The hierarchy mixes **what the actor is** with a movement rule that might change during the game. A power-up could make a walking actor fly temporarily. Changing the object's class at runtime is awkward.

Compose a movement policy instead:

PYTHON
```
class StepMovement:
    def __init__(self, distance=1):
        self.distance = distance

    def move(self, position):
        return position + self.distance


class Actor:
    def __init__(self, name, movement):
        self.name = name
        self.movement = movement
        self.position = 0

    def advance(self):
        self.position = self.movement.move(self.position)
        return self.position
```

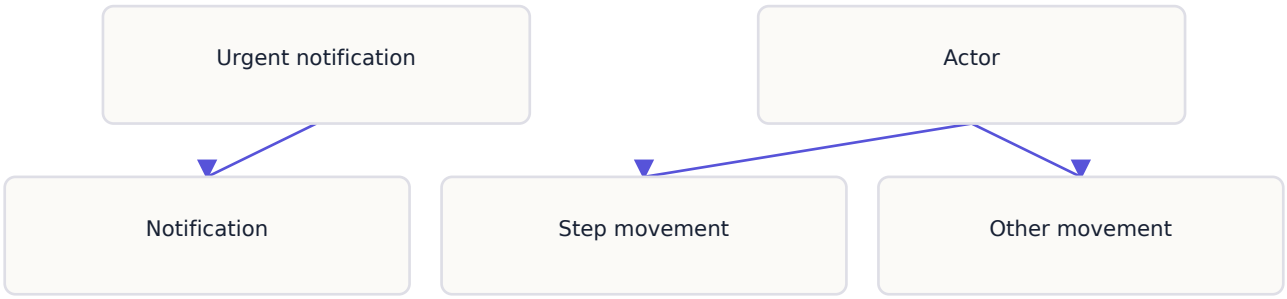
PYTHON
```
actor = Actor("Pip", StepMovement(1))
assert actor.advance() == 1

actor.movement = StepMovement(3)
assert actor.advance() == 4
```

The actor owns identity and current position. The policy owns how a position advances. This design supports a temporary change without a class explosion.

Composition is not always better. If `UrgentNotification` is a stable semantic kind of notification and preserves the contract, inheritance communicates that relationship well. Prefer composition as the default for reuse and varying behavior; accept inheritance when substitutability makes the relationship true.

Inheritance fixes a subtype relationship. Composition attaches replaceable behavior to an object that keeps its own identity.



Urgent notification → Notification: is substitutable for

Actor → Step movement: has movement policy

Actor → Other movement: policy can change

6. Use the smallest honest structure

Before creating a class, walk through this table:

Structure	Strong signal	Warning sign
function	input becomes output; no owned evolving state	hidden global mutation
dictionary/list	local, flexible data shape	many callers rely on magic keys and shared rules
dataclass	fields define value, representation, equality	unique entity is accidentally equal by fields
regular class	coherent state changes through guarded behavior	class only wraps one stateless function
composition	separate responsibilities collaborate or vary	dozens of tiny objects only forward calls
inheritance	stable subtype preserves one public contract	used only to borrow code or satisfy “is-a” wording

Watch for these smells:

- every noun in a requirement becomes a class;
- every field receives a getter and setter without a rule;
- a `Manager`, `Controller`, or `System` owns every unrelated task;
- a one-method class has no configuration, state, or replaceable behavior;

- subclasses inspect and mutate several base underscore fields;
- hierarchies grow a new subtype for every combination of features;
- a class exists only to make two lines “object-oriented.”

A module function can be a public, testable abstraction. A dictionary can be the clearest record. The aim is coherent ownership, not ceremony.

Review a design with change scenarios

Static diagrams can make almost any decomposition look tidy. Test the design by asking where realistic changes land:

1. **Add a third route strategy.** Ideally add one collaborator and selection configuration, not edit every explorer method.
2. **Change energy validation.** Ideally edit the entity that owns energy, not each planner.
3. **Display status in another language.** Ideally change a presenter or formatter, not route algorithms.
4. **Record planner calls.** Ideally decorate or compose the planner behavior, not create subclasses for every explorer-policy combination.
5. **Make a planner stateful.** Decide whether it is shared or independently constructed; do not let that lifecycle be accidental.

When one small requirement crosses most classes, either the requirement is inherently cross-cutting or responsibilities have been split along the wrong lines. Use the change scenario as evidence, not as proof that more classes are always needed.

7. Design a museum tour with replaceable route policies

Build these types:

PYTHON

```
from dataclasses import dataclass


@dataclass(frozen=True)
class Stop:
    name: str
    floor: int
    popularity: int


class FloorOrder:
    def arrange(self, stops):
        """Return stops ordered by floor, then name."""
        ...


class PopularFirst:
    def arrange(self, stops):
        """Return stops by descending popularity, then name."""
        ...


class Tour:
    def __init__(self, title, stops, policy):
        """Store a non-empty title, copied stops, and an arrangement policy."""
        ...
```

PYTHON (CONTINUED)

```
    def itinerary(self):
        """Delegate arrangement and return an independent list of stop names."""
        ...
```

Contracts:

- a stop name is non-blank, floor is non-negative, and popularity is 0–5;
- `Tour` requires at least one stop and stores its own list copy;
- each policy returns a new list without mutating the supplied collection;
- `itinerary()` delegates to the current policy;
- changing `tour.policy` changes arrangement, not tour identity or stop data; and
- no project-defined base class is needed for the two policies.

Checks:

PYTHON

```
stops = [
    Stop("Moon Engine", 2, 5),
    Stop("Clockwork Birds", 1, 3),
    Stop("Amber Maps", 1, 4),
]

tour = Tour("Museum Sprint", stops, FloorOrder())
assert tour.itinerary() == ["Amber Maps", "Clockwork Birds", "Moon Engine"]

tour.policy = PopularFirst()
assert tour.itinerary() == ["Moon Engine", "Amber Maps", "Clockwork Birds"]

assert [stop.name for stop in stops] == [
    "Moon Engine",
    "Clockwork Birds",
    "Amber Maps",
]

stops.append(Stop("New Exhibit", 0, 1))
assert "New Exhibit" not in tour.itinerary()
```

Then write a five-sentence design note:

1. Why is `Stop` a value-like dataclass?
2. Why is `Tour` a stateful regular class?
3. What behavior do both policies promise?
4. Why is composition more useful than inheriting `PopularTour(Tour)` here?
5. Which part could remain a function without harming the design?

Hint 1

Validate `Stop` in `__post_init__`. In `Tour.__init__`, use `self.stops = list(stops)` so later mutation of the caller's list does not alter the tour.

Hint 2

`FloorOrder.arrange` can use `sorted(stops, key=lambda stop: (stop.floor, stop.name))`. The popular policy can sort with `(-stop.popularity, stop.name)`.

Hint 3

`Tour.itinerary` asks `self.policy.arrange(self.stops)` and converts the returned stops to names. It should not use `isinstance` to choose an algorithm; the collaborator owns that decision.

Compare a complete composed tour

PYTHON

```
from dataclasses import dataclass


@dataclass(frozen=True)
class Stop:
    name: str
    floor: int
    popularity: int

    def __post_init__(self):
        cleaned = self.name.strip()
        if not cleaned:
            raise ValueError("name cannot be blank")
        if self.floor < 0:
            raise ValueError("floor cannot be negative")
        if not 0 <= self.popularity <= 5:
            raise ValueError("popularity must be between 0 and 5")
        object.__setattr__(self, "name", cleaned)


class FloorOrder:
    def arrange(self, stops):
        return sorted(stops, key=lambda stop: (stop.floor, stop.name))


class PopularFirst:
    def arrange(self, stops):
        return sorted(stops, key=lambda stop: (-stop.popularity, stop.name))


class Tour:
    def __init__(self, title, stops, policy):
        cleaned = title.strip()
        copied_stops = list(stops)
        if not cleaned:
            raise ValueError("title cannot be blank")
        if not copied_stops:
            raise ValueError("stops cannot be empty")
        self.title = cleaned
        self.stops = copied_stops
        self.policy = policy

    def itinerary(self):
        arranged = self.policy.arrange(self.stops)
        return [stop.name for stop in arranged]


stops = [
    Stop("Moon Engine", 2, 5),
    Stop("Clockwork Birds", 1, 3),
    Stop("Amber Maps", 1, 4),
]
tour = Tour("Museum Sprint", stops, FloorOrder())
assert tour.itinerary() == ["Amber Maps", "Clockwork Birds", "Moon Engine"]
```

PYTHON (CONTINUED)

```
tour.policy = PopularFirst()
assert tour.itinerary() == ["Moon Engine", "Amber Maps", "Clockwork Birds"]
```

The two policies happen to be stateless and could be functions passed into the tour. The small policy classes are justified here because they make a named, replaceable behavior visible; a callable-function design would also be honest.

Checkpoint: make the design choice explicit

Choose the smallest honest abstraction

Choose the structure that best matches the stated responsibility.

1. A tour may switch between floor order and popularity order at runtime. Which design fits best?

- **A.** Create a deep subclass for every possible combination.
- **B.** Compose the tour with a replaceable arrangement policy.
- **C.** Store the algorithm in a global string.
- **D.** Duplicate the complete Tour class.

2. A stateless operation converts Celsius to Fahrenheit. What is the best first structure?

- **A.** Five inherited converter classes
- **B.** A mutable singleton
- **C.** A function with an explicit input and return value
- **D.** A manager object with getters

3. Why is inheriting only to reuse three implementation lines risky?

- **A.** Inheritance prevents methods.
- **B.** Python forbids short base classes.
- **C.** Composition cannot reuse behavior.
- **D.** Inheritance declares a substitutable public relationship that code reuse alone does not justify.

Write your choices before checking the answer key at the end of this book.

8. Key points for collaborating objects

- Split classes by coherent responsibilities and reasons to change, not an arbitrary method-count limit.
- Composition lets an object hold or receive another object. Delegation asks that collaborator to perform the behavior it owns.
- Constructor injection makes dependencies visible and lets callers supply a deterministic or alternative implementation.
- Duck typing relies on behavioral contracts: accepted inputs, results, failures, and side effects—not just matching method names.

- Inheritance states that a subtype can be used wherever callers expect the base contract. An “is-a” sentence alone does not prove that promise.
- Use `super()` to continue cooperative lookup and preserve base initialization or behavior in an honest shallow hierarchy.
- A subtype that rejects valid base input, weakens results, or breaks postconditions is not safely substitutable.
- Prefer composition for varying capabilities and code reuse. Accept inheritance when the semantic subtype and behavioral contract are stable.
- Functions, collections, and dataclasses remain valid alternatives. A class should earn its state ownership or collaboration role.

References

- Python tutorial: Inheritance
- Python glossary: Duck typing
- Python built-in `super`
- PEP 8: Designing for inheritance
- Think Python: Classes and methods

Model a Complete Road-Crossing Game

Online lesson and runnable notebook

Grow explicit road-crossing rules into a deterministic model with value objects, protected entities, replaceable traffic policies, turn results, and no hidden input or randomness.

- **Level:** Python Foundations
- **Estimated time:** 5–6 hours
- **You will learn:** Translate game rules into acceptance examples, assign value/entity/policy/orchestrator responsibilities, process deterministic turns, return immutable snapshots, and place a changed rule in its proper owner.
- **Practice in:** Google Colab, JupyterLab, or a local Python interpreter

A player begins at the bottom of a five-column board. Reaching the top row wins. Vehicles move horizontally after each accepted player move; landing on the same position as a vehicle loses.

That description is enough to imagine a game but not enough to implement one. Does traffic move before or after the player? Does an invalid direction consume a turn? Do vehicles wrap at the board edge? Can a finished game receive another move? A dependable object model begins by making those rules observable.

This case study combines the unit’s ideas:

- frozen values represent positions, vehicles, and turn reports;
- stateful objects own player and lane transitions;
- a traffic policy varies behavior without inheritance;
- a game composes the participants and owns turn order; and
- input, printing, animation, timing, and randomness stay outside the rules.

1. Turn the story into acceptance examples

Use this board convention:

- rows run from `0` at the start to `goal_row` at the finish;
- columns run from `0` through `width - 1`;
- directions are `"up"`, `"down"`, `"left"`, and `"right"`;
- a move outside the board is rejected before anything changes;
- after an accepted player move, every lane advances;
- vehicles wrap horizontally;
- collision is checked after traffic advances;
- collision takes priority over winning if a future rule permits both; and
- a finished game rejects later turns.

Write representative transitions before choosing classes:

Starting state	Request	Traffic after turn	Result
player `(0, 2)`, vehicle `(1, 4)`	`up`	vehicle wraps to `(1, 0)`	continue at `(1, 2)`
player `(0, 2)`, vehicle `(1, 1)`	`up`	vehicle reaches `(1, 2)`	collision and loss
player `(0, 0)`	`left`	unchanged	reject; no turn consumed
player `(2, 2)`, goal row 3	`up`	lanes advance	win at `(3, 2)`
game already won	any direction	unchanged	reject finished game

Acceptance examples answer design arguments later. If collision results differ, inspect the agreed operation order instead of randomly moving code.

2. Let the procedural sketch reveal its pressure points

A dictionary version can process a simple move:

```python
game = {
    "width": 5,
    "goal_row": 3,
    "player": {"row": 0, "column": 2},
    "vehicles": [{"row": 1, "column": 4, "speed": 1}],
    "status": "running",
    "turn": 0,
}


def step_up(state):
    state["player"]["row"] += 1
    for vehicle in state["vehicles"]:
        vehicle["column"] = (
            vehicle["column"] + vehicle["speed"]
        ) % state["width"]
    state["turn"] += 1
```

This is a reasonable spike: it proves wrapping arithmetic. It is not yet the complete contract. Any caller can insert a vehicle on the wrong row, set a negative width, misspell `"column"`, call after completion, or mutate the player without advancing traffic.

Do not replace the dictionary merely because it looks “procedural.” Identify the responsibilities that have lasting rules:

Concept	Role	Why
position	frozen value	row and column determine its meaning
direction	plain string plus a small lookup	no independent identity or behavior needed
player	stateful entity	owns current position and valid movement

Concept	Role	Why
vehicle	frozen value	advancing can produce another vehicle value
lane	stateful entity	owns vehicles and applies one traffic policy
speed policy	collaborator	determines movement for a turn and can vary
turn result	frozen value	records observable evidence from one turn
game	orchestrator	owns ordering, status, collision, and win rules

This list also records concepts that **do not** become classes. A direction can remain a string constrained by a dictionary. A small helper can flatten vehicle positions. Resisting unnecessary classes is part of the design.

Checkpoint: classify the concepts

Assign each game responsibility

Choose the structure whose semantics match the stated concept.

1. Why is a frozen `Position(row, column)` a useful value object?

- **A.** Its fields determine the coordinate, and movement can produce another position without changing the original.
- **B.** It reads keyboard input automatically.
- **C.** It must own every lane.
- **D.** It needs a deep subclass tree.

2. Which responsibility most clearly belongs to `Game`?

- **A.** Formatting every terminal color
- **B.** Choosing a random keyboard
- **C.** Coordinating player movement, traffic advancement, collision, and result order
- **D.** Implementing integer addition

3. Why can `"up"` remain a plain string in this foundation model?

- **A.** Strings can inherit from Game.
- **B.** A small lookup fully maps the finite request to row/column changes; no separate identity is needed.
- **C.** Plain values cannot be validated.
- **D.** Every game must avoid classes.

Write your choices before checking the answer key at the end of this book.

3. Build immutable coordinates and vehicle values

Define a coordinate that rejects negative values and returns new positions:

```python
from dataclasses import dataclass, replace


@dataclass(frozen=True)
class Position:
    row: int
    column: int

    def __post_init__(self):
        if self.row < 0 or self.column < 0:
            raise ValueError("position values cannot be negative")

    def shifted(self, row_change=0, column_change=0):
        return Position(
            self.row + row_change,
            self.column + column_change,
        )
```

`shifted` does not mutate the source:

```python
start = Position(0, 2)
next_position = start.shifted(row_change=1)

assert start == Position(0, 2)
assert next_position == Position(1, 2)
```

Represent a vehicle as another value. It owns horizontal wrapping because that rule determines the next vehicle value:

```python
@dataclass(frozen=True)
class Vehicle:
    position: Position

    def advanced(self, distance, width):
        if distance < 0:
            raise ValueError("distance cannot be negative")
        if width <= 0:
            raise ValueError("width must be positive")
        next_column = (self.position.column + distance) % width
        return replace(
            self,
            position=Position(self.position.row, next_column),
        )
```

```python
vehicle = Vehicle(Position(1, 4))
moved = vehicle.advanced(distance=1, width=5)

assert moved == Vehicle(Position(1, 0))
assert vehicle == Vehicle(Position(1, 4))
```

Returning a new vehicle makes before/after reasoning straightforward. A large real-time game might choose mutable vehicle entities for performance, but that need does not exist here.

4. Give the player one valid movement entrance

The player has identity and an evolving position, so use a regular class:

```python
DIRECTION_CHANGES = {
    "up": (1, 0),
    "down": (-1, 0),
    "left": (0, -1),
    "right": (0, 1),
}


class Player:
    def __init__(self, name, position, *, width, goal_row):
        cleaned = name.strip()
        if not cleaned:
            raise ValueError("name cannot be blank")
        if width <= 0 or goal_row <= 0:
            raise ValueError("board dimensions must be positive")
        if not 0 <= position.column < width:
            raise ValueError("player column is outside the board")
        if not 0 <= position.row <= goal_row:
            raise ValueError("player row is outside the board")
        self.name = cleaned
        self._position = position
        self._width = width
        self._goal_row = goal_row

    @property
    def position(self):
        return self._position

    def preview(self, direction):
        try:
            row_change, column_change = DIRECTION_CHANGES[direction]
        except KeyError as error:
            raise ValueError(f"unknown direction: {direction}") from error

        candidate = self._position.shifted(row_change, column_change)
        if not 0 <= candidate.column < self._width:
            raise ValueError("move leaves the board columns")
        if not 0 <= candidate.row <= self._goal_row:
            raise ValueError("move leaves the board rows")
        return candidate

    def move(self, direction):
        candidate = self.preview(direction)
        self._position = candidate
        return candidate
```

`preview` validates and calculates without changing state. `move` commits that candidate. The split lets `Game` validate the player request before the rest of the turn begins, while the player remains the owner of board-boundary rules.

```python
player = Player("Pip", Position(0, 2), width=5, goal_row=3)
assert player.preview("up") == Position(1, 2)
assert player.position == Position(0, 2)
assert player.move("up") == Position(1, 2)
```

Try an invalid boundary and preserve state:

```python
edge = Player("Edge", Position(0, 0), width=5, goal_row=3)

try:
    edge.move("left")
except ValueError as error:
    print(error)

assert edge.position == Position(0, 0)
```

5. Let a lane own traffic and receive its speed policy

Start with two policies that share behavior but no project-defined parent:

```python
class FixedSpeed:
    def __init__(self, distance):
        if distance < 0:
            raise ValueError("distance cannot be negative")
        self.distance = distance

    def distance_for(self, turn):
        return self.distance


class AlternatingSpeed:
    def __init__(self, first, second):
        if first < 0 or second < 0:
            raise ValueError("distances cannot be negative")
        self.first = first
        self.second = second

    def distance_for(self, turn):
        return self.first if turn % 2 == 1 else self.second
```

A lane ensures every vehicle stays on its row and delegates speed selection:

```python
class Lane:
    def __init__(self, row, width, vehicles, speed_policy):
        if row <= 0:
            raise ValueError("lane row must be positive")
        if width <= 0:
            raise ValueError("width must be positive")
        copied = list(vehicles)
        if any(vehicle.position.row != row for vehicle in copied):
            raise ValueError("every vehicle must be on the lane row")
        if any(vehicle.position.column >= width for vehicle in copied):
            raise ValueError("vehicle column is outside the lane")
        self.row = row
        self.width = width
        self._vehicles = copied
        self._speed_policy = speed_policy

    @property
    def vehicles(self):
        return tuple(self._vehicles)

    def advance(self, turn):
        distance = self._speed_policy.distance_for(turn)
        if distance < 0:
            raise ValueError("policy returned a negative distance")
        self._vehicles = [
            vehicle.advanced(distance, self.width)
            for vehicle in self._vehicles
```

PYTHON (CONTINUED)

```
        ]
        return self.vehicles
```

Returning a tuple keeps callers from appending to the lane's internal list:

PYTHON

```
lane = Lane(
    row=1,
    width=5,
    vehicles=[Vehicle(Position(1, 4))],
    speed_policy=FixedSpeed(1),
)

assert lane.advance(turn=1) == (Vehicle(Position(1, 0)),)
assert lane.advance(turn=2) == (Vehicle(Position(1, 1)),)
```

`Lane` does not ask `isinstance(policy, FixedSpeed)`. It uses the `distance_for(turn)` behavior. An alternating policy can replace the fixed one without changing lane code.

6. Return turn evidence instead of printing inside the model

The game needs to communicate what happened. Printing from `step` would force every caller to parse text and mix presentation with rules. Use a frozen result:

PYTHON

```
@dataclass(frozen=True)
class TurnResult:
    turn: int
    status: str
    player: Position
    traffic: tuple[Position, ...]
    message: str
```

The game can return this value to a notebook, command-line interface, web page, or graphical renderer. Those callers may present it differently while the model's evidence remains the same.

Build the orchestrator:

PYTHON

```
class Game:
    def __init__(self, player, lanes, *, goal_row):
        copied_lanes = list(lanes)
        rows = [lane.row for lane in copied_lanes]
        if len(rows) != len(set(rows)):
            raise ValueError("lane rows must be unique")
        if any(not 0 < row < goal_row for row in rows):
            raise ValueError("lanes must be between start and goal")
        self.player = player
        self.lanes = copied_lanes
        self.goal_row = goal_row
        self.turn = 0
        self.status = "running"

    def _traffic_positions(self):
        return tuple(
            vehicle.position
            for lane in self.lanes
            for vehicle in lane.vehicles
        )

    def snapshot(self, message="Ready"):
```

PYTHON (CONTINUED)

```
        return TurnResult(
            turn=self.turn,
            status=self.status,
            player=self.player.position,
            traffic=self._traffic_positions(),
            message=message,
        )

    def step(self, direction):
        if self.status != "running":
            raise RuntimeError("game is already finished")

        self.player.preview(direction)
        self.turn += 1
        self.player.move(direction)

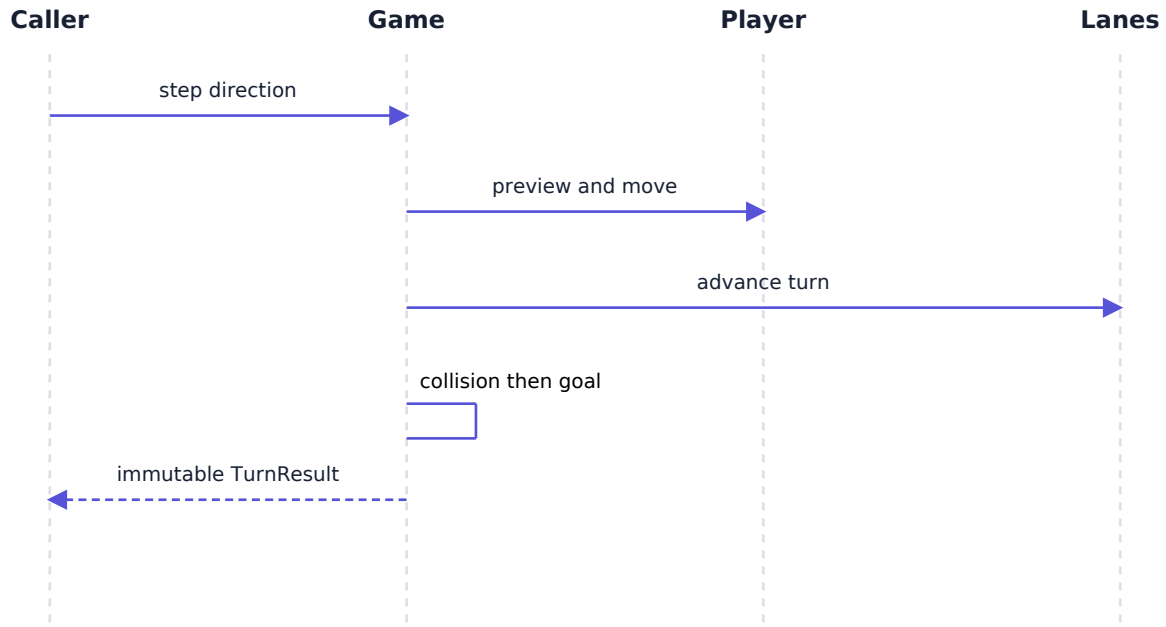
        for lane in self.lanes:
            lane.advance(self.turn)

        traffic = self._traffic_positions()
        if self.player.position in traffic:
            self.status = "lost"
            message = "A vehicle reached the player."
        elif self.player.position.row == self.goal_row:
            self.status = "won"
            message = "The player reached the goal."
        else:
            message = "The crossing continues."

        return self.snapshot(message)
```

Why call `preview` and then `move`, which previews again? The first call rejects an invalid direction before the game increments its turn. The second asks the player to commit through its own public operation. In a larger model, `move` could accept the validated candidate or `Game` could use a command object. The small duplication is clear and keeps ownership intact here.

`Game` owns cross-object order. Each participant owns its local operation and the result captures evidence for the caller.



Checkpoint: trace one complete turn

Follow the deterministic turn order

Use the stated move-then-traffic-then-collision contract.

1. The player moves to `(1, 2)` and a vehicle advances from `(1, 1)` to `(1, 2)`. What is the result?

- **A.** The game ignores the shared position.
- **B.** The player wins immediately.
- **C.** Traffic moves back.
- **D.** The game reports a collision and becomes lost.

2. A player at column 0 requests `left`. What should happen to the turn and traffic?

- **A.** Both advance anyway.
- **B.** The request is rejected before the turn increments or traffic moves.
- **C.** The player wraps to column 4.
- **D.** The game becomes won.

3. Why does `step` return a frozen `TurnResult` instead of printing?

- **A.** Callers can inspect stable evidence and choose their own presentation.
- **B.** Frozen values read keyboard input faster.
- **C.** Printing is not valid Python.
- **D.** A result automatically saves to disk.

Write your choices before checking the answer key at the end of this book.

7. Run a safe turn, a collision, and a win

Build a helper so each scenario gets fresh objects:

```python
def make_game(vehicle_column=4):
    player = Player("Pip", Position(0, 2), width=5, goal_row=3)
    lane = Lane(
        row=1,
        width=5,
        vehicles=[Vehicle(Position(1, vehicle_column))],
        speed_policy=FixedSpeed(1),
    )
    return Game(player, [lane], goal_row=3)
```

Safe first turn:

```python
game = make_game(vehicle_column=4)
result = game.step("up")

assert result.turn == 1
assert result.player == Position(1, 2)
assert result.traffic == (Position(1, 0),)
```

PYTHON (CONTINUED)

```
assert result.status == "running"
```

Trace that call in order:

1. `Game.step("up")` confirms the game is running.
2. `Player.preview("up")` maps the direction to `(1, 0)`, derives `Position(1, 2)`, and proves it lies on the board without mutation.
3. The game increments its turn from zero to one.
4. `Player.move("up")` commits the same valid coordinate.
5. `Lane.advance(1)` asks `FixedSpeed` for distance one.
6. The vehicle at column four produces a new vehicle at column zero through modulo wrapping.
7. The player at `(1, 2)` is absent from the traffic tuple, so collision does not change status.
8. Row one is below goal row three, so the game remains running.
9. `snapshot` returns final player and traffic values in a frozen result.

This trace distinguishes the call stack from the domain sequence. The player and lane perform local operations, while the game decides their order. If a future animation wants to draw intermediate frames, it can receive additional events without moving presentation into these classes.

Collision after traffic:

PYTHON

```
collision_game = make_game(vehicle_column=1)
result = collision_game.step("up")

assert result.player == Position(1, 2)
assert result.traffic == (Position(1, 2),)
assert result.status == "lost"
```

Three safe upward turns reach the goal:

PYTHON

```
winning_game = make_game(vehicle_column=4)

first = winning_game.step("up")
second = winning_game.step("up")
third = winning_game.step("up")

assert first.status == "running"
assert second.status == "running"
assert third.status == "won"
assert third.player == Position(3, 2)

try:
    winning_game.step("left")
except RuntimeError as error:
    assert str(error) == "game is already finished"
```

The scenarios create independent games. Reusing one finished game’s player or lane in another game would share mutable participants. The constructor accepts objects deliberately, so the caller owns that decision. A production model could reject reused ownership or construct from immutable configuration.

The independence boundary deserves an explicit check:

```python
first_game = make_game(vehicle_column=4)
second_game = make_game(vehicle_column=4)

first_game.step("up")

assert first_game.player.position == Position(1, 2)
assert second_game.player.position == Position(0, 2)
assert first_game.lanes[0] is not second_game.lanes[0]
```

`make_game` constructs a fresh player, lane, list, vehicle, and policy each time. If a helper cached and reused a mutable lane, game instances would not be independent even though their `Game` objects had different identities.

8. Repair ownership mistakes before adding features

Global player state

If `player_position` is global, two games silently share it. Put position on each `Player` instance and give each game its own player.

Rendering inside `Player.move`

If `move` prints a board, a web caller and terminal caller cannot choose different views. Return position/result evidence. Presentation belongs outside.

`VehiclePlayer` inheritance for shared movement code

A vehicle is not substitutable for a player: it wraps horizontally according to a lane policy, while a player accepts directions and respects board edges. Extract coordinate arithmetic into `Position.shifted` and keep the two domain contracts separate.

Random speed inside `Lane.advance`

Hidden randomness makes the same input produce different output. Inject a policy or precomputed sequence. A graphical version may use a random policy, while checks use a deterministic one.

9. Add a shelter row without scattering conditions

Suppose row 1 contains a marked shelter. Traffic still advances through it, but the player cannot collide while standing on that row. This rule coordinates the player's final position with traffic, so `Game` owns it.

Extend construction with a copied immutable set:

```python
# Inside Game.__init__ after validating lane rows:
# self.shelter_rows = frozenset(shelter_rows)
# if any(not 0 < row < goal_row for row in self.shelter_rows):
#     raise ValueError("shelter rows must be between start and goal")
```

Change only the collision condition:

```python
# if (
#     self.player.position.row not in self.shelter_rows
#     and self.player.position in traffic
# ):
#     self.status = "lost"
```

Do not teach vehicles to inspect shelters; vehicles still move normally. Do not teach `Player` to inspect every lane; it still owns board-bounded movement. The size and location of the change are evidence that responsibilities are coherent.

10. Complete the model and changed rule

Recreate the complete model using these required names:

- `Position`, `Vehicle`, `Player`, `FixedSpeed`, `AlternatingSpeed`, `Lane`, `TurnResult`, and `Game`;
- `Position.shifted`, `Vehicle.advanced`, `Player.preview`, `Player.move`, `Lane.advance`, `Game.snapshot`, and `Game.step`; and
- `shelter_rows=()` as an optional keyword in `Game.__init__`.

Add these checks to the earlier scenarios:

```python
player = Player("Pip", Position(0, 2), width=5, goal_row=3)
lane = Lane(
    1,
    5,
    [Vehicle(Position(1, 1))],
    FixedSpeed(1),
)
sheltered = Game(player, [lane], goal_row=3, shelter_rows={1})

result = sheltered.step("up")
assert result.player == Position(1, 2)
assert result.traffic == (Position(1, 2),)
assert result.status == "running"

before = sheltered.snapshot()
try:
    sheltered.step("sideways")
except ValueError:
    pass
after = sheltered.snapshot()

assert after == before
```

Then explain:

1. why `Position`, `Vehicle`, and `TurnResult` use value equality;
2. why `Player`, `Lane`, and `Game` own changing identity;
3. why traffic speed is injected rather than selected with `isinstance`;
4. why direction remains a simple value;
5. why the shelter rule belongs to `Game`; and
6. how a CLI or graphical view could use `TurnResult` without changing rules.

Hint 1

Build and check in dependency order: values, player, policies, lane, result, then game. Do not debug the whole object graph before its small parts pass.

Hint 2

`Game.step` should preview first, increment the turn, commit player movement, advance each lane, flatten traffic positions, check non-sheltered collision, then check the goal.

Hint 3

Store `shelter_rows` as a `frozenset` after verifying each row lies strictly between start and goal. Include no shelter details in `Player` or `Lane`; only the game's collision condition changes.

Compare the complete game model after running your scenarios

PYTHON

```
from dataclasses import dataclass, replace


DIRECTION_CHANGES = {
    "up": (1, 0),
    "down": (-1, 0),
    "left": (0, -1),
    "right": (0, 1),
}


@dataclass(frozen=True)
class Position:
    row: int
    column: int

    def __post_init__(self):
        if self.row < 0 or self.column < 0:
            raise ValueError("position values cannot be negative")

    def shifted(self, row_change=0, column_change=0):
        return Position(self.row + row_change, self.column + column_change)


@dataclass(frozen=True)
class Vehicle:
    position: Position

    def advanced(self, distance, width):
        if distance < 0:
            raise ValueError("distance cannot be negative")
        if width <= 0:
            raise ValueError("width must be positive")
        column = (self.position.column + distance) % width
        return replace(self, position=Position(self.position.row, column))


class Player:
    def __init__(self, name, position, *, width, goal_row):
        cleaned = name.strip()
        if not cleaned:
            raise ValueError("name cannot be blank")
        if width <= 0 or goal_row <= 0:
            raise ValueError("board dimensions must be positive")
        if not 0 <= position.column < width:
            raise ValueError("player column is outside the board")
        if not 0 <= position.row <= goal_row:
            raise ValueError("player row is outside the board")
```

PYTHON (CONTINUED)

```
        self.name = cleaned
        self._position = position
        self._width = width
        self._goal_row = goal_row

    @property
    def position(self):
        return self._position

    def preview(self, direction):
        try:
            row_change, column_change = DIRECTION_CHANGES[direction]
        except KeyError as error:
            raise ValueError(f"unknown direction: {direction}") from error
        candidate = self._position.shifted(row_change, column_change)
        if not 0 <= candidate.column < self._width:
            raise ValueError("move leaves the board columns")
        if not 0 <= candidate.row <= self._goal_row:
            raise ValueError("move leaves the board rows")
        return candidate

    def move(self, direction):
        self._position = self.preview(direction)
        return self._position


class FixedSpeed:
    def __init__(self, distance):
        if distance < 0:
            raise ValueError("distance cannot be negative")
        self.distance = distance

    def distance_for(self, turn):
        return self.distance


class AlternatingSpeed:
    def __init__(self, first, second):
        if first < 0 or second < 0:
            raise ValueError("distances cannot be negative")
        self.first = first
        self.second = second

    def distance_for(self, turn):
        return self.first if turn % 2 == 1 else self.second


class Lane:
    def __init__(self, row, width, vehicles, speed_policy):
        if row <= 0:
            raise ValueError("lane row must be positive")
        if width <= 0:
            raise ValueError("width must be positive")
        copied = list(vehicles)
        if any(vehicle.position.row != row for vehicle in copied):
            raise ValueError("every vehicle must be on the lane row")
        if any(vehicle.position.column >= width for vehicle in copied):
            raise ValueError("vehicle column is outside the lane")
        self.row = row
        self.width = width
        self._vehicles = copied
        self._speed_policy = speed_policy

    @property
    def vehicles(self):
        return tuple(self._vehicles)

    def advance(self, turn):
```

PYTHON (CONTINUED) (CONTINUED)

```
        distance = self._speed_policy.distance_for(turn)
        if distance < 0:
            raise ValueError("policy returned a negative distance")
        self._vehicles = [
            vehicle.advanced(distance, self.width)
            for vehicle in self._vehicles
        ]
        return self.vehicles


@dataclass(frozen=True)
class TurnResult:
    turn: int
    status: str
    player: Position
    traffic: tuple[Position, ...]
    message: str


class Game:
    def __init__(self, player, lanes, *, goal_row, shelter_rows=()):
        copied_lanes = list(lanes)
        rows = [lane.row for lane in copied_lanes]
        if len(rows) != len(set(rows)):
            raise ValueError("lane rows must be unique")
        if any(not 0 < row < goal_row for row in rows):
            raise ValueError("lanes must be between start and goal")
        shelters = frozenset(shelter_rows)
        if any(not 0 < row < goal_row for row in shelters):
            raise ValueError("shelter rows must be between start and goal")
        self.player = player
        self.lanes = copied_lanes
        self.goal_row = goal_row
        self.shelter_rows = shelters
        self.turn = 0
        self.status = "running"

    def _traffic_positions(self):
        return tuple(
            vehicle.position
            for lane in self.lanes
            for vehicle in lane.vehicles
        )

    def snapshot(self, message="Ready"):
        return TurnResult(
            self.turn,
            self.status,
            self.player.position,
            self._traffic_positions(),
            message,
        )

    def step(self, direction):
        if self.status != "running":
            raise RuntimeError("game is already finished")
        self.player.preview(direction)
        self.turn += 1
        self.player.move(direction)
        for lane in self.lanes:
            lane.advance(self.turn)
        traffic = self._traffic_positions()
        if (
            self.player.position.row not in self.shelter_rows
            and self.player.position in traffic
        ):
            self.status = "lost"
            message = "A vehicle reached the player."
```

PYTHON (CONTINUED) (CONTINUED) (CONTINUED)

```
        elif self.player.position.row == self.goal_row:
            self.status = "won"
            message = "The player reached the goal."
        else:
            message = "The crossing continues."
        return self.snapshot(message)
```

Run the safe, collision, win, invalid-direction, and shelter scenarios against this definition. The result records make every transition inspectable without a renderer.

Checkpoint: place a changed rule

Change the model without tangling it

Choose the owner whose responsibility already contains the rule.

1. A shelter prevents collision on selected rows while traffic still moves. Which object should apply that cross-object rule?

- **A.** `Vehicle`, because vehicles should inspect every player
- **B.** The direction string
- **C.** `Game`, where collision between player and traffic is already decided
- **D.** A global print function

2. Why should `Vehicle` not inherit from `Player` merely to reuse coordinate movement?

- **A.** Frozen values cannot have methods.
- **B.** Players cannot have positions.
- **C.** Inheritance always uses global state.
- **D.** Their movement contracts differ; shared coordinate arithmetic belongs in `Position` or a helper.

3. What is the clearest way to add variable traffic speed while keeping exact scenarios repeatable?

- **A.** Inject a policy with `distance_for(turn)` behavior.
- **B.** Call uncontrolled randomness inside every vehicle.
- **C.** Read keyboard input from `Lane.advance`.
- **D.** Create one global speed shared by all games.

Write your choices before checking the answer key at the end of this book.

11. Review why each class exists

- `Position` is a frozen value because row and column determine equality.
- `Vehicle` is a frozen value in this small model; advancing returns a new value and leaves before/after evidence intact.
- `Player` is a stateful entity because one named participant owns a changing position and board-bound movement.

- `Lane` is a stateful entity because it owns a collection of current vehicle values and applies one supplied traffic policy.
- `FixedSpeed` and `AlternatingSpeed` are collaborators selected through behavior, not subclasses created for ceremony.
- `TurnResult` is an immutable event record that separates rules from views.
- `Game` orchestrates one turn and owns rules that cross participant boundaries: operation order, status, collision, shelter, and victory.
- Directions remain simple strings because a small validated lookup fully expresses their role.

12. Key points from the case study

- Write rules and acceptance examples before arguing about classes.
- Let an initial procedural sketch expose pressure points; do not refactor only to make the program look object-oriented.
- Assign each concept value, entity, collaborator, orchestrator, collection, or function semantics deliberately.
- Keep domain transitions deterministic. Inject variable policies instead of hiding randomness or input inside the model.
- Validate an accepted turn before changing state, then make its operation order explicit.
- Return immutable evidence so callers can render, log, or assert behavior without reaching into internal fields.
- Do not use inheritance to share code between objects with different public contracts.
- A changed rule belongs with the object that already owns the affected decision. Small focused changes are evidence of coherent responsibilities.

References

- Python `dataclasses`
- Python tutorial: Class and instance variables
- Python glossary: Duck typing
- Python tutorial: Errors and exceptions
- Think Python: Classes and functions

Unit Challenge: Run the Tiny Robot Tournament

Online lesson and runnable notebook

Build immutable moves and results, robots with protected energy and shield state, and an arena whose atomic turns include a playful combo spotlight.

- **Level:** Python Foundations · Unit 11 challenge
- **Estimated time:** 60–120 minutes
- **Outcome assessed:** Choose value objects, stateful entities, and composition deliberately; preserve valid robot state; process rejected turns atomically; and evolve arena-owned history without adding an unnecessary hierarchy.
- **Evidence:** Thirty progressive assertions, a complete two-robot tournament, one changed rule, and a short responsibility map

1. Enter the Tiny Robot Tournament

The workshop lights dim. **Bolt** and **Nova** roll into a tabletop arena, each with an energy meter and a shield. A move costs the acting robot energy and reduces the opponent's shield. Accepted turns alternate between the two robots. The first robot to reduce the other shield to zero wins.

Your model must answer every turn with data, not print statements. A notebook, CLI, or graphical interface can decide later how to announce the result.

The challenge is deliberately smaller than a commercial battle system. There are no inventories, teams, animations, artificial intelligence, or networked players. Those features would hide the object-design decisions beneath unrelated work. The fun comes from making a compact rules engine behave exactly as the thirty checks predict.

After the base tournament works, add the **combo spotlight**: when the same robot successfully repeats the same named move on its next turn, that move applies one extra point of impact. Rejected turns do not change the active robot, spend energy, damage a shield, or update combo history.

You do not need inheritance. `Arena` composes two `Robot` objects and uses immutable `Move` and `TurnResult` values. Choosing not to create `AttackRobot`, `DefendRobot`, or `ComboRobot` subclasses is part of the design.

Advance one green check at a time

Implement the types in dependency order: `Move`, `Robot`, `TurnResult`, then `Arena`. Run the check harness after each stage. The first failure names the earliest contract that still needs work. :::

2. Translate the tournament rules into contracts

`Move` is an immutable value

- Strip surrounding whitespace from `name` and reject a blank result.
- `energy_cost` and `impact` are non-negative integers; booleans do not count as integers for this model.

- A move cannot have both zero cost and zero impact.
- Two moves with the same normalized field values compare equal.
- Ordinary field assignment after construction is rejected.

`Robot` owns valid changing state

- Strip `name`; reject blank names.
- Initial `energy` and `shield` are integers from 1 through 20.
- Expose read-only `energy`, `shield`, and `active` properties.
- `spend_energy(amount)` accepts a non-negative integer no greater than current energy, then subtracts it.
- `absorb_impact(amount)` accepts a non-negative integer and reduces shield with a floor at zero.
- Every rejected operation leaves the robot unchanged.
- A robot is active exactly while shield is greater than zero.

`TurnResult` is immutable evidence

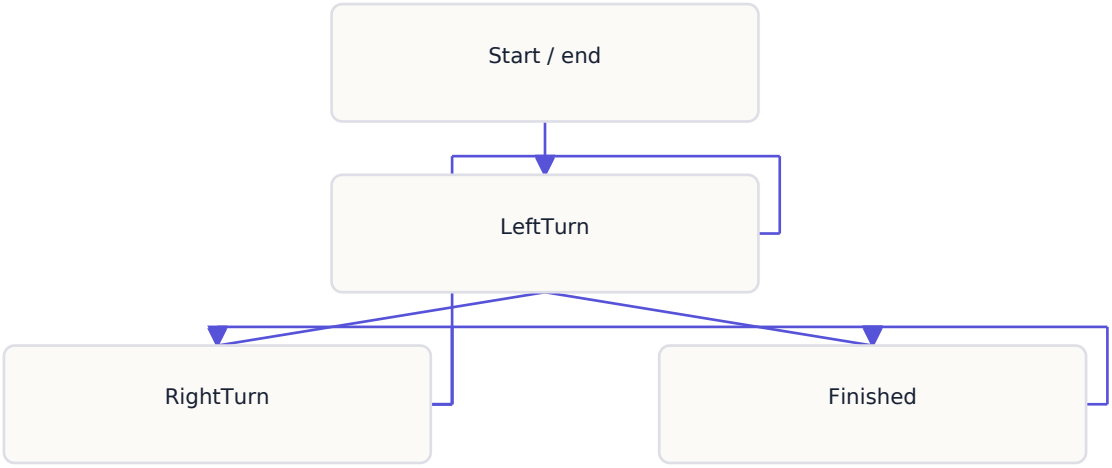
Record:

- actor and target names;
- normalized move name;
- applied impact, including any combo point;
- whether the spotlight combo activated;
- actor energy after spending;
- target shield after impact; and
- status: `"running"` or `"finished"`.

`Arena` coordinates the workflow

- Receive two different robot objects with different names.
- The left robot acts first; accepted turns alternate.
- Expose the current robot as a read-only property.
- `winner` is `None` while both shields are active and the winning robot’s name after completion.
- Reject a turn after completion.
- Reject a move the actor cannot afford before changing any object or history.
- On success: calculate combo, spend energy, apply impact, record history, advance the active side, and return `TurnResult`.
- Combo history belongs to the arena because it compares successful turns across time; neither `Move` nor `Robot` needs tournament-specific history.

Rejected requests return to the same state. Only an accepted turn can update robots, history, and the current side.



LeftTurn → LeftTurn: unaffordable move rejected

LeftTurn → RightTurn: accepted move and both active

RightTurn → RightTurn: unaffordable move rejected

RightTurn → LeftTurn: accepted move and both active

LeftTurn → Finished: accepted move lowers shield to zero

RightTurn → Finished: accepted move lowers shield to zero

Finished → Finished: later turn rejected

3. Start from the contract

Copy this starter into one cell or Python file. Replace each `...` without renaming the public classes, properties, or methods.

PYTHON

```
from dataclasses import dataclass


@dataclass(frozen=True)
class Move:
    """A named tournament move with an energy cost and shield impact."""

    name: str
    energy_cost: int
    impact: int

    def __post_init__(self):
        ...


class Robot:
    """A tournament robot that protects its energy and shield state."""

    def __init__(self, name, *, energy=10, shield=10):
        ...

    @property
    def energy(self):
        ...

    @property
    def shield(self):
```

PYTHON (CONTINUED)

```
        ...

    @property
    def active(self):
        ...

    def spend_energy(self, amount):
        """Spend an affordable non-negative amount and return energy left."""
        ...

    def absorb_impact(self, amount):
        """Reduce shield to no less than zero and return shield left."""
        ...


@dataclass(frozen=True)
class TurnResult:
    """An immutable report of one accepted tournament turn."""

    actor: str
    target: str
    move: str
    impact: int
    combo: bool
    energy_left: int
    shield_left: int
    status: str


class Arena:
    """Coordinate two robots and the rules for accepted tournament turns."""

    def __init__(self, left, right):
        ...

    @property
    def current_robot(self):
        ...

    @property
    def winner(self):
        ...

    def play_turn(self, move):
        """Apply one affordable move atomically and return its result."""
        ...
```

Do not add `input`, `print`, randomness, files, or global tournament state to these classes. You may add private helper methods or attributes when they make one responsibility clearer.

4. Build one responsibility at a time

Work through four short stages:

1. **Move values:** normalize and validate `Move`, then prove equality, hashability, and frozen assignment before touching robot state.
2. **Robot invariants:** construct one robot, spend energy, absorb impact, and prove every rejected operation preserves the before snapshot.
3. **Base arena:** compose two fresh robots, process one turn from each side, reject unaffordable work, and finish a tournament.

4 **Combo spotlight:** remember the last successful move independently for each robot, then prove rejected work never enters that history.

After a failure, keep the smallest stage in view. A broken `Move` invariant cannot be repaired by changing turn alternation, and an incorrect winner should not lead you to rewrite dataclass equality.

5. Run progressive assertions

Run this harness below your implementation. It deliberately uses ordinary assertions rather than pytest.

PYTHON

```
from dataclasses import FrozenInstanceError


checks_run = 0


def check(condition, message):
    """Count one passing tournament contract or raise its message."""
    global checks_run
    assert condition, message
    checks_run += 1


def raises(exception_type, action):
    """Return whether calling action raises the expected exception type."""
    try:
        action()
    except exception_type:
        return True
    return False


# Move contracts: checks 1–7
pulse = Move(" Pulse ", energy_cost=2, impact=3)
check(pulse == Move("Pulse", 2, 3), "Move should normalize and compare by value")
check(pulse in {Move("Pulse", 2, 3)}, "a fully hashable frozen Move should hash")


def rename_pulse():
    pulse.name = "Other"


check(raises(FrozenInstanceError, rename_pulse), "Move should be frozen")
check(raises(ValueError, lambda: Move("   ", 1, 1)), "blank move should fail")
check(raises(ValueError, lambda: Move("Bad", -1, 1)), "negative cost should fail")
check(raises(ValueError, lambda: Move("Bad", 1, -1)), "negative impact should fail")
check(raises(ValueError, lambda: Move("Nap", 0, 0)), "a move must do something")


# Robot contracts: checks 8–15
probe = Robot(" Bolt ", energy=10, shield=8)
check(
    (probe.name, probe.energy, probe.shield) == ("Bolt", 10, 8),
    "Robot should normalize its name and expose initial state",
)
check(probe.active is True, "a positive shield should be active")
check(probe.spend_energy(3) == 7, "spending should return remaining energy")
check(
    probe.absorb_impact(20) == 0 and probe.active is False,
    "impact should floor shield at zero and deactivate the robot",
)
probe_state = (probe.energy, probe.shield)
check(
    raises(ValueError, lambda: probe.spend_energy(8)),
```

PYTHON (CONTINUED)

```
    "overspending should fail",
)
check((probe.energy, probe.shield) == probe_state, "failed spending should be atomic")
check(
    raises(ValueError, lambda: probe.absorb_impact(-1)),
    "negative impact should fail",
)


def assign_energy():
    probe.energy = 99


check(raises(AttributeError, assign_energy), "energy should be read-only")


# Arena and combo contracts: checks 16–30
solo = Robot("Solo")
check(raises(ValueError, lambda: Arena(solo, solo)), "one object cannot fill both slots")
check(
    raises(ValueError, lambda: Arena(Robot("Echo"), Robot(" Echo "))),
    "robot names must differ",
)

left = Robot("Bolt", energy=8, shield=8)
right = Robot("Nova", energy=8, shield=8)
arena = Arena(left, right)
check(arena.current_robot is left and arena.winner is None, "left should act first")

jab = Move("Jab", energy_cost=2, impact=3)
first = arena.play_turn(jab)
check(
    first == TurnResult("Bolt", "Nova", "Jab", 3, False, 6, 5, "running"),
    "first turn result is wrong",
)
check(
    (left.energy, right.shield, arena.current_robot) == (6, 5, right),
    "first turn state or alternation is wrong",
)


def rewrite_result():
    first.combo = True


check(raises(FrozenInstanceError, rewrite_result), "TurnResult should be frozen")

spark = Move("Spark", energy_cost=1, impact=2)
second = arena.play_turn(spark)
check(
    second == TurnResult("Nova", "Bolt", "Spark", 2, False, 7, 6, "running"),
    "second robot turn is wrong",
)

before_rejection = (
    left.energy,
    left.shield,
    right.energy,
    right.shield,
    arena.current_robot,
)
check(
    raises(ValueError, lambda: arena.play_turn(Move("Nova Beam", 99, 1)))
    and before_rejection
    == (
        left.energy,
        left.shield,
        right.energy,
```

PYTHON (CONTINUED) (CONTINUED)

```
        right.shield,
        arena.current_robot,
    ),
    "an unaffordable turn should change nothing",
)

third = arena.play_turn(jab)
check(
    third == TurnResult("Bolt", "Nova", "Jab", 4, True, 4, 1, "running"),
    "Bolt's repeated move should receive the combo point",
)
fourth = arena.play_turn(spark)
check(
    fourth == TurnResult("Nova", "Bolt", "Spark", 3, True, 6, 3, "running"),
    "Nova's repeated move should track its own combo",
)
final = arena.play_turn(jab)
check(
    final == TurnResult("Bolt", "Nova", "Jab", 4, True, 2, 0, "finished")
    and arena.winner == "Bolt"
    and right.active is False,
    "finishing turn or winner is wrong",
)

finished_state = (left.energy, left.shield, right.energy, right.shield, arena.winner)
check(
    raises(RuntimeError, lambda: arena.play_turn(Move("Tap", 0, 1)))
    and finished_state
    == (left.energy, left.shield, right.energy, right.shield, arena.winner),
    "a finished arena should reject later turns without mutation",
)

other_left = Robot("Tin", energy=4, shield=4)
other_right = Robot("Copper", energy=4, shield=4)
other_arena = Arena(other_left, other_right)
check(
    other_arena.current_robot is other_left
    and (other_left.energy, other_right.shield) == (4, 4),
    "a separate arena should own independent participants and turn state",
)

low = Robot("Low", energy=1, shield=4)
steady = Robot("Steady", energy=4, shield=4)
rejection_arena = Arena(low, steady)
rejection_state = (low.energy, steady.shield, rejection_arena.current_robot)
check(
    raises(ValueError, lambda: rejection_arena.play_turn(Move("Heavy", 2, 2)))
    and rejection_state
    == (low.energy, steady.shield, rejection_arena.current_robot),
    "rejection should not advance state or current robot",
)
tap = rejection_arena.play_turn(Move("Tap", 0, 1))
check(
    tap.combo is False and tap.impact == 1 and rejection_arena.current_robot is steady,
    "a rejected move must not create combo history",
)

print(f"Tournament ready: {checks_run} checks passed")
```

Expected final output:

TEXT

```
Tournament ready: 30 checks passed
```

Checks 1–7: move values

- ☐ names normalize and equal fields compare equal;
- ☐ an all-hashable frozen move can participate in a set;
- ☐ field assignment is rejected;
- ☐ blank name is rejected;
- ☐ negative energy cost is rejected;
- ☐ negative impact is rejected; and
- ☐ a zero-cost, zero-impact move is rejected.

Checks 8–15: robot invariants

- ☐ name and initial state are normalized/exposed;
- ☐ positive shield means active;
- ☐ spending returns remaining energy;
- ☐ impact floors shield at zero and updates active state;
- ☐ overspending raises;
- ☐ failed spending preserves state;
- ☐ negative impact raises; and
- ☐ public energy assignment is unavailable.

Checks 16–30: arena composition and history

- ☐ one robot object cannot occupy both slots;
- ☐ names must identify different competitors;
- ☐ the left robot begins;
- ☐ first result has exact field values;
- ☐ first mutation and alternation are correct;
- ☐ results are frozen;
- ☐ the right robot’s first turn is correct;
- ☐ an unaffordable move is atomic;
- ☐ Bolt’s repeat earns its own combo;
- ☐ Nova’s repeat earns a separate combo;
- ☐ the finishing result, winner, and inactive target agree;
- ☐ finished play is rejected atomically;
- ☐ another arena remains independent;
- ☐ rejection does not advance the current robot; and
- ☐ rejection does not create combo history.

6. Use the hint ladder only when needed

Hint 1

Map responsibilities before filling methods:

```text
Move and TurnResult -> frozen field-based values
Robot               -> validates and owns energy/shield mutation
Arena               -> owns left/right order, winner, and last move per robot
```

Normalize frozen `Move.name` in `__post_init__` with `object.__setattr__(self, "name", cleaned)`. Store robot state in `_energy` and `_shield`, then expose read-only properties.

Hint 2

In `play_turn`, choose actor and target from a two-item tuple. Before mutation, check completion and affordability. A dictionary keyed by each robot object can store its last successful normalized move name. Combo is `last_moves.get(actor) == move.name`.

Do not write history or change the active index until validation has passed.

Hint 3

The successful order is:

1. reject a finished arena;
2. select actor and target;
3. reject an unaffordable move;
4. calculate `combo` and `applied_impact`;
5. call `actor.spend_energy`;
6. call `target.absorb_impact`;
7. remember the actor's move;
8. alternate the current index;
9. derive status from `winner`; and
10. return all final values in `TurnResult`.

Clamp shield inside `Robot.absorb_impact` with `max(0, ...)`.

7. Keep debugging evidence

Record one failure that improved your understanding:

Evidence	Your record
Earliest failing check	number and exact assertion message
Before state	actor energy, both shields, current robot, and known winner
Expected contract	one sentence describing what should happen

Evidence	Your record
Hypothesis	the specific validation, ordering, or ownership rule involved
Controlled repair	one method or responsibility changed
After state	passing check plus one nearby regression check

For example, if an unaffordable turn switches the current robot, the defect is not in `Robot.spend_energy`: the arena advanced its index before completing validation. If a repeated move never gains impact, inspect when and under which robot key successful history is stored. Preserve the failing observation rather than replacing it with “it did not work.”

8. Compare a complete tournament model

Show the complete implementation after attempting all stages

PYTHON

```
from dataclasses import dataclass


def require_non_negative_integer(value, label):
    """Validate one tournament count."""
    if not isinstance(value, int) or isinstance(value, bool):
        raise TypeError(f"{label} must be an integer")
    if value < 0:
        raise ValueError(f"{label} cannot be negative")


@dataclass(frozen=True)
class Move:
    """A named tournament move with an energy cost and shield impact."""

    name: str
    energy_cost: int
    impact: int

    def __post_init__(self):
        if not isinstance(self.name, str):
            raise TypeError("name must be a string")
        cleaned = self.name.strip()
        if not cleaned:
            raise ValueError("name cannot be blank")
        require_non_negative_integer(self.energy_cost, "energy_cost")
        require_non_negative_integer(self.impact, "impact")
        if self.energy_cost == 0 and self.impact == 0:
            raise ValueError("a move must cost energy or apply impact")
        object.__setattr__(self, "name", cleaned)


class Robot:
    """A tournament robot that protects its energy and shield state."""

    def __init__(self, name, *, energy=10, shield=10):
        if not isinstance(name, str):
            raise TypeError("name must be a string")
        cleaned = name.strip()
        if not cleaned:
            raise ValueError("name cannot be blank")
        require_non_negative_integer(energy, "energy")
        require_non_negative_integer(shield, "shield")
        if not 1 <= energy <= 20:
            raise ValueError("energy must be between 1 and 20")
```

PYTHON (CONTINUED)

```
        if not 1 <= shield <= 20:
            raise ValueError("shield must be between 1 and 20")
        self.name = cleaned
        self._energy = energy
        self._shield = shield

    @property
    def energy(self):
        return self._energy

    @property
    def shield(self):
        return self._shield

    @property
    def active(self):
        return self._shield > 0

    def spend_energy(self, amount):
        require_non_negative_integer(amount, "amount")
        if amount > self._energy:
            raise ValueError("not enough energy")
        self._energy -= amount
        return self._energy

    def absorb_impact(self, amount):
        require_non_negative_integer(amount, "amount")
        self._shield = max(0, self._shield - amount)
        return self._shield


@dataclass(frozen=True)
class TurnResult:
    """An immutable report of one accepted tournament turn."""

    actor: str
    target: str
    move: str
    impact: int
    combo: bool
    energy_left: int
    shield_left: int
    status: str


class Arena:
    """Coordinate two robots and the rules for accepted tournament turns."""

    def __init__(self, left, right):
        if not isinstance(left, Robot) or not isinstance(right, Robot):
            raise TypeError("left and right must be Robot instances")
        if left is right:
            raise ValueError("left and right must be different objects")
        if left.name == right.name:
            raise ValueError("robot names must differ")
        self._robots = (left, right)
        self._current_index = 0
        self._last_moves = {}

    @property
    def current_robot(self):
        return self._robots[self._current_index]

    @property
    def winner(self):
        left, right = self._robots
        if not left.active:
            return right.name
```

PYTHON (CONTINUED) (CONTINUED)

```
        if not right.active:
            return left.name
        return None

    def play_turn(self, move):
        if not isinstance(move, Move):
            raise TypeError("move must be a Move")
        if self.winner is not None:
            raise RuntimeError("tournament is already finished")

        actor = self._robots[self._current_index]
        target = self._robots[1 - self._current_index]

        if move.energy_cost > actor.energy:
            raise ValueError("not enough energy for this move")

        combo = self._last_moves.get(actor) == move.name
        applied_impact = move.impact + (1 if combo else 0)

        actor.spend_energy(move.energy_cost)
        target.absorb_impact(applied_impact)
        self._last_moves[actor] = move.name
        self._current_index = 1 - self._current_index

        status = "finished" if self.winner is not None else "running"
        return TurnResult(
            actor=actor.name,
            target=target.name,
            move=move.name,
            impact=applied_impact,
            combo=combo,
            energy_left=actor.energy,
            shield_left=target.shield,
            status=status,
        )
```

Run the unchanged thirty-check harness below this implementation. Notice the responsibility boundaries:

- the validation helper owns a repeated primitive rule, not object state;
- `Move` and `TurnResult` get field equality and frozen behavior;
- `Robot` is a regular class because it owns guarded mutation and identity; and
- `Arena` composes robots and owns tournament order/history.

9. Explain the combo change

The base battle needs current robot, energy, shield, and winner. Combo adds one new question: “Which successful move did this same actor use on its previous personal turn?”

That history belongs to `Arena` because:

- a move value should describe one move, not remember who used it;
- a robot may participate in a future game with different tournament rules;
- the arena already owns alternation and accepted-turn order; and
- rejected turns must not enter successful tournament history.

The dictionary uses robot objects as keys. Regular `Robot` instances keep identity-based hashing and equality because the class does not define `__eq__`. Two robots named Bolt are still different objects, although this arena also rejects duplicate display names to keep results unambiguous.

Try one more transfer variation after all checks pass:

1. add a third move named `"Guard Tap"` with zero energy cost and one impact;
2. confirm the first use is not a combo;
3. let the other robot take an accepted turn;
4. repeat `"Guard Tap"` and confirm impact becomes two; and
5. replace it with a different move on the next personal turn and confirm the combo resets.

This modification needs no subclass. It changes arena-owned history and result evidence while robot invariants remain untouched.

10. Draw the responsibility map

Complete this short record in your own words:

Question	Your explanation
Which types are value objects?	
Which type is a stateful entity?	
Which type orchestrates collaborators?	
Which function remains a plain function, and why?	
Why is inheritance unnecessary?	
Which fields must remain unchanged after rejection?	
Why does combo history belong to the arena?	

A strong explanation names observable contracts. “Arena is the manager” is too vague. “Arena owns which robot acts, successful move history, winner calculation, and the order of atomic mutation” is specific enough to review.

11. Check your understanding

Tiny Robot Tournament design check

Connect every answer to the tournament’s ownership and state contracts.

1. Which object should reject spending more energy than is available?

- **A.** The immutable result
- **B.** The `Robot` that owns current energy
- **C.** The move name string
- **D.** A global print function

2. An actor cannot afford a move. What must remain unchanged?

- **A.** Both robots, current side, winner, and combo history
- **B.** Only the move’s display name
- **C.** The target changes but the actor does not
- **D.** The current side advances anyway

3. Why does combo history belong to `Arena`?

- **A.** Frozen dataclasses cannot contain text.
- **B.** Every robot must inherit Arena.
- **C.** History is always global.
- **D.** The arena owns accepted turn order and compares each robot’s successful moves across turns.

4. Why is `Move` a frozen dataclass?

- **A.** It needs hidden changing identity.
- **B.** Its normalized fields define a stable value with useful equality and representation.
- **C.** It must read keyboard input.
- **D.** It controls both robots.

5. Why does the challenge not need `ComboRobot(Robot)`?

- **A.** Python classes cannot inherit.
- **B.** Combos require files.
- **C.** Combo is arena-owned turn history; ordinary composed robots already honor the needed state behavior.
- **D.** Subclasses cannot have energy.

Write your choices before checking the answer key at the end of this book.

12. Record the challenge result

Record completion after:

- all thirty checks pass in one clean run;
- your implementation contains no input, printing, global tournament state, or unnecessary project-defined inheritance;
- an unaffordable move and a finished-game move both preserve prior state;
- both competitors can earn separate combos;
- the result values exactly report base and combo impact;
- the additional `Guard Tap` variation behaves as predicted; and
- your responsibility map explains the choices in behavioral terms.

This button stores progress only in this browser. It does not submit code, grade identity, or issue a certificate.

Key points

- Immutable dataclasses fit move descriptions and accepted-turn evidence because their declared fields determine value semantics.
- A regular robot class owns identity and guarded mutation. Read-only properties expose state without offering arbitrary replacement.
- The arena composes two robots and owns the rules that cross them: active side, affordability-before-commit, winner, and successful-move history.
- Rejected operations are atomic: no energy, shield, current side, winner, or combo fact changes.
- The combo rule belongs to tournament history and can evolve without a robot subclass.
- Progressive assertions let a learner locate the earliest broken contract and continue without a testing framework.

Unit 12: Command-Line Applications

Online lesson and runnable notebook

Learn how Unit 12 turns reusable Python into a discoverable, configurable command whose output, diagnostics, logs, and exit status form a dependable public interface.

- **Level:** Python Foundations
- **Estimated time:** 45–60 minutes
- **You will learn:** Identify the parts of a dependable command-line interface, prepare notebook and terminal workspaces, and plan the evidence you will collect across five connected lessons.
- **Practice in:** Google Colab or JupyterLab for direct calls, plus a local terminal for real command behavior

Imagine that a trail team receives status lines from several checkpoints:

```text
north-ridge|open|Windy above the tree line
river-gate|closed|Bridge inspection in progress
```

The transformation code already works as a Python function. A command-line application makes that behavior useful outside one notebook. A teammate can ask for help, read a file or pipe data into the program, select an operation, and send the result into another program:

```bash
$ trail-report --help
usage: trail-report [-h] {summarize,check} ...

$ cat trail.txt | trail-report check - --format jsonl
{"checkpoint": "north-ridge", "status": "open"}
trail-report: line 2: closed checkpoints require attention
$ echo $?
1
```

Four observations are visible in this short session:

1. help describes a **command grammar** rather than forcing the user to inspect Python source;
2. standard input lets the command receive another program's output;
3. JSON data goes to stdout while the explanation goes to stderr; and
4. the exit status gives a calling shell or automation tool a small, machine-readable result.

A dependable CLI is therefore more than "a script that reads arguments." It is a boundary between a person or another process and reusable application logic. Every visible behavior at that boundary is part of the command's interface.

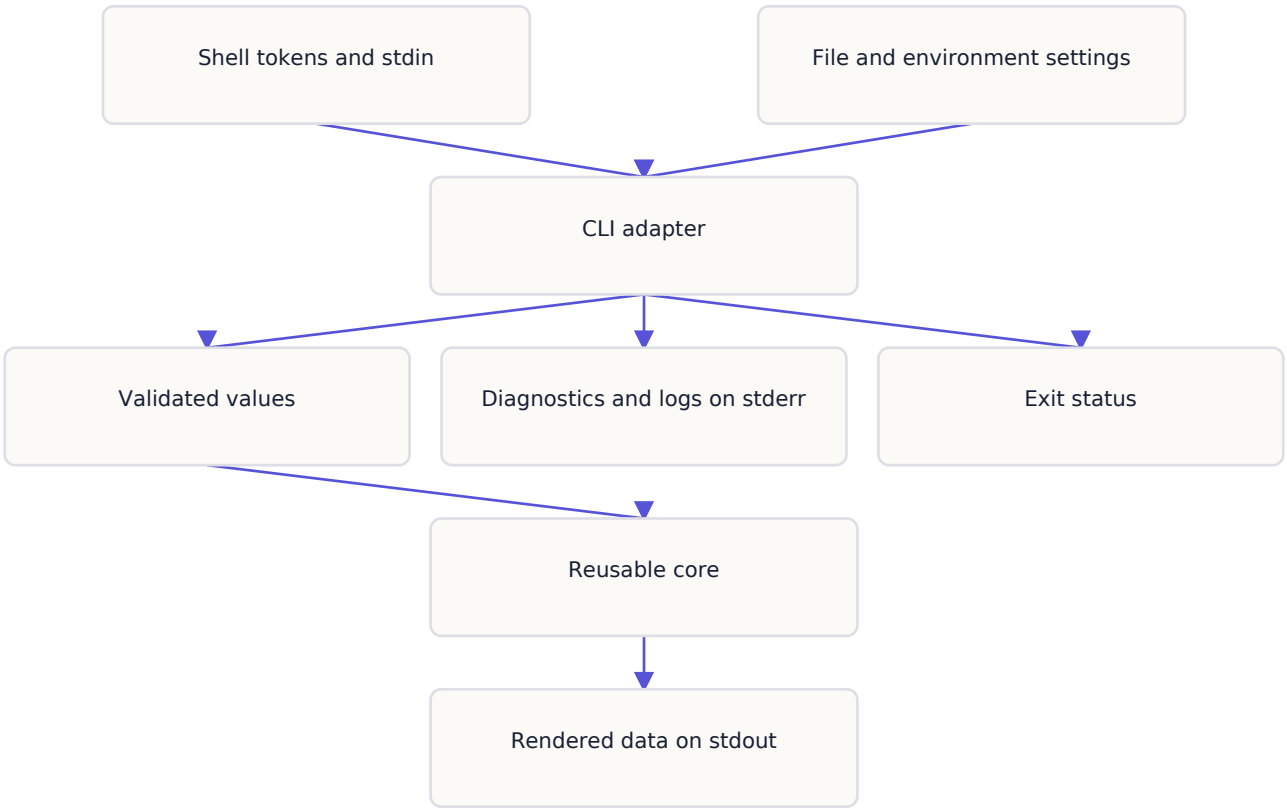
What you will be able to build

By the end of Unit 12, you will be able to:

- turn a written command contract into positionals, options, flags, and subcommands with `argparse`;

- explain how shell tokens become `sys.argv` strings and parsed Python values;
- keep structured data on stdout and useful diagnostics or logs on stderr;
- return an application status from `main()` and let the entry point turn it into a process exit status;
- resolve defaults, a TOML file, environment variables, and explicit CLI values in a documented order;
- validate settings once and avoid ordinary secret-disclosure mistakes;
- configure logging at the application boundary without contaminating data;
- keep core functions independent of argparse namespaces and global streams;
- inspect behavior through direct calls and through a real child process; and
- decide what argparse, Click, Typer, and Rich do—and what they do not repair.

Notice that the command adapter translates between process-facing details and ordinary Python values; the reusable core does not need to know about the shell.



The five-stage trail-report build

The lessons add one boundary at a time. The same trail-report scenario makes the dependencies visible, but each page includes a complete snapshot so that a missed lesson fragment does not leave broken glue code.

Step	Lesson	What changes in the command	Evidence you keep
1	Design a Friendly Command with argparse	`summarize` and check gain clear arguments, generated help, validation, and dispatch.	Example invocations, namespace values, help, and one usage error.
2	Put Output, Errors, and Exit Status in the Right Place	The command accepts stdin, produces JSON Lines, reports rejected records separately, and returns a documented status.	Exact stdout, stderr, and status from direct and process calls.
3	Resolve Configuration Without Exposing Secrets	Defaults, TOML, environment, and CLI values become one validated settings object.	A precedence trace, source labels, invalid-setting evidence, and a redacted secret.
4	Write Useful Logs Without Polluting Output	Quiet and verbose modes add application-owned operational records on stderr.	Captured records at several levels and unchanged stdout.
5	Keep the Command Thin and the Core Reusable	Parser, settings, core, rendering, boundary, and entry point become a small package.	Direct checks plus a real process launched from another directory.

The unit challenge then changes both the story and data. You will build **Decode the Deep-Sea Beacon**, a streaming command that decodes messages, resolves its shift, handles damaged pings, supports strict mode, and remains useful in a pipeline.

Bring forward what you already know

Unit 12 combines earlier skills rather than starting a separate kind of Python:

- **Unit 9:** paths, encodings, context managers, JSON, and validation at an external-data boundary;
- **Unit 10:** modules, packages, `__main__`, console entry points, virtual environments, and installed-project checks;
- **Unit 11:** small dataclasses, cohesive responsibilities, composition, and explicit collaborators.

You do not need to remember every detail. Keep those units available for reference. This unit will say when a task intentionally reuses them.

Unit 13 will teach pytest and the design of maintainable test suites. Here, we use plain assertions, `StringIO`, and one `subprocess.run()` smoke check because we need immediate evidence while building a CLI. Those checks are useful, but they are not presented as a substitute for the testing unit.

Use two workspaces for two kinds of evidence

A notebook is excellent for direct calls

In Colab or JupyterLab, you can pass argument lists explicitly:

```python
arguments = ["check", "signals.txt", "--status", "open"]
print(arguments)
```

You can also use `io.StringIO` as a small in-memory text stream. This makes stdout and stderr easy to inspect without launching another process.

A terminal reveals the process boundary

A local terminal is the best place to observe:

- how your shell splits or quotes arguments;
- pipes and redirects;
- `python -m package` and an installed console command;
- stdout and stderr as separate channels;
- the process exit status; and
- behavior when the current directory is not the project directory.

Create a disposable practice directory. A virtual environment is useful but no third-party CLI library is required:

```bash
mkdir unit12-cli-lab
cd unit12-cli-lab
python --version
python -c "import argparse, logging, tomllib; print('standard library ready')"
```

On Windows PowerShell, `py` may be the Python launcher and `$LASTEXITCODE` reports the most recent native process status. On POSIX-like shells, `python3` may be the command and `$?` reports the status. Use the spelling that works in your environment; the Python program itself should not depend on one shell.

A notebook cell is not a shell command

Text such as `trail-report --help` is shell syntax. Calling `parser.parse_args(["--help"])` is a Python function call. Both are valuable, but they pass through different boundaries. Each lesson labels which one you are observing. :::

Keep a CLI evidence notebook

For each lesson, record at least one row in this table:

Invocation or call	Expected stdout	Expected stderr	Expected status/exception	Actual result
`trail-report check good.txt`	one JSON object	empty	0	
`trail-report check mixed.txt`	valid objects only	rejected line	1	
`parse_args(["check", "--help"])`	help captured by the parser	empty	`SystemExit(0)`	

The exact output matters. “It failed” is weak evidence; the channel, message, and status explain what contract was observed. Restart a notebook and run from the top before declaring a direct-call example reproducible. Run process checks from outside the project before declaring an entry point installable.

Plan the active work

Budget approximately **22–30 hours** for the unit, including typing examples, predicting behavior, checkpoints, repair clinics, lesson labs, and the challenge. A useful pace is:

1. stop after the command-grammar lab;
2. take a second session for streams and process status;
3. split configuration and secrets across two shorter sessions;
4. practice logging in a fresh interpreter so old handlers do not hide cause and effect;
5. give the architecture lab its own long session; and
6. attempt the challenge later, without copying the trail-report solution.

You are ready to continue when you can point to the adapter, core, stdout, stderr, and status in the opening diagram—even if you cannot implement all of them yet.

Begin by designing the public command

Continue to Design a Friendly Command with argparse. Keep this overview open when deciding whether a new piece belongs to the process boundary or the reusable core. :::

Design a Friendly Command with argparse

Online lesson and runnable notebook

Turn example terminal invocations into a clear argparse grammar with positionals, options, flags, validation, generated help, and focused subcommands.

- **Level:** Python Foundations
- **Estimated time:** 4–5 hours
- **You will learn:** Specify a command grammar, explain shell-to-argv conversion, choose clear argument shapes, validate command syntax, read generated help, and dispatch focused subcommands.
- **Practice in:** Google Colab or JupyterLab for parser calls, plus a local terminal for real shell tokenization

A trail coordinator wants one program to summarize reports and check them for a particular status. These two invocations are the design sketch:

```bash
trail-report summarize signals.txt --limit 5
trail-report check - --status closed --tag weather --tag safety
```

Before writing Python, a user can already infer much of the interface:

- `summarize` and `check` select different jobs;
- an input path follows the job, and `-` probably means standard input;
- `--limit` and `--status` attach named values;
- `--tag` may be repeated; and
- ordering and spelling are intentional.

That public shape is a **command grammar**. `argparse` translates valid strings in that grammar into Python values, generates help, and rejects invalid shapes consistently.

As you work, answer these questions:

- How does text typed in a shell become a Python list of argument strings?
- When is a positional, option, flag, repeatable option, or subcommand clearest?
- Which invalid values should argparse reject before the core runs?
- What can help and usage errors tell both a user and a developer?
- How can a parser be inspected without changing the notebook’s real arguments?

1. Read the command before writing the parser

Annotate this command one token at a time:

```text
trail-report check reports/today.txt --status closed --verbose
```

Token	Role	Expected Python meaning
`trail-report`	program name	identifies the installed command
`check`	subcommand	selects the check operation
`reports/today.txt`	positional value	input path
`--status`	option	names the next token's purpose
`closed`	option value	text, later constrained to known statuses
`--verbose`	flag	`True` when present

The table is more useful than immediately calling `add_argument()`. It answers which values are essential, which have defaults, and which combinations make sense. Add three more examples before implementation:

```bash
trail-report summarize signals.txt
trail-report summarize signals.txt --limit 10
trail-report check - --status open --tag river --tag bridge
```

Then list invalid shapes deliberately:

```text
trail-report summarize
trail-report summarize signals.txt --limit zero
trail-report check signals.txt --status maybe
```

A good parser should reject all three with usage evidence. The reusable trail logic should never need to interpret `"zero"` as an integer or guess what `"maybe"` means.

2. Follow shell text into `sys.argv`

The shell and Python do different jobs. The shell first turns a command line into argument strings. It starts the process and supplies those strings. Python exposes them as `sys.argv`:

```python
import sys

for index, value in enumerate(sys.argv):
    print(index, repr(value))
```

When a file named `show_args.py` contains that code, a shell session might be:

```bash
$ python show_args.py check "North Ridge.txt" --limit 3
0 'show_args.py'
1 'check'
2 'North Ridge.txt'
3 '--limit'
4 '3'
```

Quotes kept `North Ridge.txt` together as one argument. The quote characters themselves are not normally part of the value. Every element is a string, including `"3"`. `argparse` can perform the requested integer conversion later.

Different shells have different quoting and variable-expansion rules. Do not attempt to reproduce a shell by calling `.split()` on a string:

PYTHON
```
command_text = 'check "North Ridge.txt" --limit 3'
print(command_text.split())
```

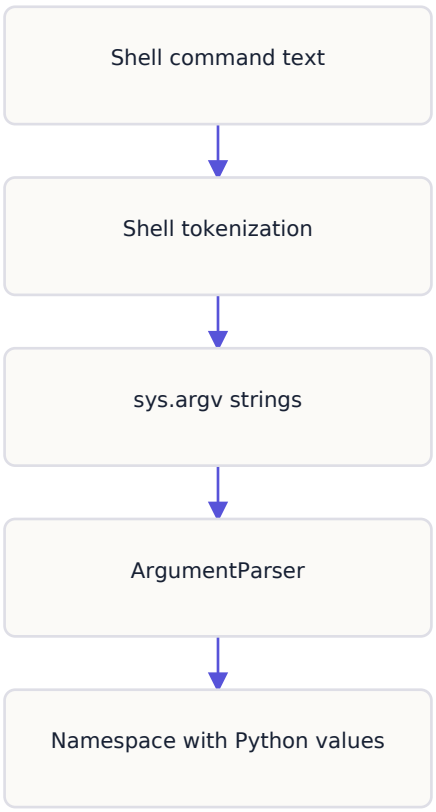
TEXT
```
['check', '"North', 'Ridge.txt"', '--limit', '3']
```

That list contains the quotes and splits the path incorrectly. In parser labs, write the already-tokenized list explicitly:

PYTHON
```
example_argv = ["check", "North Ridge.txt", "--limit", "3"]
print(example_argv)
```

`argparse.ArgumentParser.parse_args()` accepts such a list. When no list is passed, it reads the real process arguments after `sys.argv[0]`. Explicit lists make notebook experiments predictable.

Notice that quoting belongs to the shell step; argparse receives a list of strings and returns a namespace containing converted values.



Use `--` when a value resembles an option

Suppose a file is literally named `--draft`. The parser may interpret it as an option. Many command grammars use a standalone `--` to end option processing:

PYTHON
```
import argparse

parser = argparse.ArgumentParser(prog="show-input")
parser.add_argument("input")
namespace = parser.parse_args(["--", "--draft"])
print(namespace.input)
```

TEXT
```
--draft
```

This is not a universal promise that every application must accept every filename. It is a widely understood boundary worth preserving when argparse's normal grammar supports it.

Checkpoint: from shell tokens to namespace values

From shell tokens to namespace values

Choose the best explanation for each observed value.

1. Why can "North Ridge.txt" appear as one sys.argv element after a normal shell invocation?

- **A.** The shell uses the quotes to group the characters before starting Python
- **B.** argparse joins every pair of words
- **C.** Python file paths always ignore spaces
- **D.** sys.argv stores the entire command in one string

2. A user types –limit 3. What reaches argparse before conversion?

- **A.** The integer 3
- **B.** The string "3"
- **C.** A float created by the shell
- **D.** No value because options cannot contain numbers

3. What is the purpose of a standalone – before –draft?

- **A.** It comments out the rest of the command
- **B.** It turns every later value into an integer
- **C.** It ends option processing so –draft can be positional data
- **D.** It requests verbose help

Write your choices before checking the answer key at the end of this book.

3. Let argparse write useful help

Start with the smallest honest parser:

```python
import argparse


def build_basic_parser():
    parser = argparse.ArgumentParser(
        prog="trail-report",
        description="Summarize and validate trail checkpoint reports.",
        epilog="Use '-' as INPUT to read from standard input.",
    )
    parser.add_argument("input", metavar="INPUT", help="report file or '-' for stdin")
    return parser


parser = build_basic_parser()
arguments = parser.parse_args(["signals.txt"])
print(arguments)
```

```text
Namespace(input='signals.txt')
```

The namespace is a simple object whose attributes correspond to parser destinations. `arguments.input` is still text because no conversion was requested.

Generated help is part of the interface. In a terminal, argparse prints it and exits successfully:

```bash
$ trail-report --help
usage: trail-report [-h] INPUT

Summarize and validate trail checkpoint reports.

positional arguments:
  INPUT       report file or '-' for stdin

options:
  -h, --help  show this help message and exit

Use '-' as INPUT to read from standard input.
```

Exact section labels can vary slightly by supported Python version. The stable design questions are more important: Can a new user tell what the command does? Does each metavariable communicate the expected kind of value? Does the help mention a special value such as `-`?

You can capture help in a direct check. `--help` raises `SystemExit(0)` after writing to stdout:

```python
import contextlib
import io

help_output = io.StringIO()

try:
    with contextlib.redirect_stdout(help_output):
        parser.parse_args(["--help"])
except SystemExit as error:
    help_status = error.code

print(help_status)
print("report file" in help_output.getvalue())
```

```text
0
True
```

Catch `SystemExit` here because observing help is the focused task. Do not wrap an entire application in `except SystemExit:` and discard the result; that would prevent argparse from fulfilling its process contract.

4. Choose an argument shape that matches the decision

Positionals identify the central subject

An input path is a good positional when almost every operation needs it:

```python
parser.add_argument("input", metavar="INPUT")
```

Positionals are concise, but their meaning depends on order. Five unexplained positionals make a command difficult to read. Move secondary choices behind named options.

Options attach a value to a visible name

```python
parser.add_argument(
    "--limit",
    type=int,
    default=20,
    metavar="COUNT",
    help="show at most COUNT records (default: 20)",
)
```

`type=int` converts the string during parsing. `default=20` is already an integer. The result is predictable:

```python
limit_parser = argparse.ArgumentParser(prog="limit-demo")
limit_parser.add_argument("--limit", type=int, default=20)

print(limit_parser.parse_args([]).limit)
print(limit_parser.parse_args(["--limit", "3"]).limit)
```

```text
20
3
```

Flags represent a switch

`store_true` makes absence and presence readable:

```python
flag_parser = argparse.ArgumentParser(prog="flag-demo")
flag_parser.add_argument("--strict", action="store_true")

print(flag_parser.parse_args([]).strict)
print(flag_parser.parse_args(["--strict"]).strict)
```

```text
False
True
```

Do not use `type=bool`. `bool("false")` is `True` because any non-empty string is truthy. A flag needs an action, or an option needs an explicit text-to-boolean converter.

Repeatable options collect several values

```python
tag_parser = argparse.ArgumentParser(prog="tag-demo")
tag_parser.add_argument("--tag", action="append", default=[])
tags = tag_parser.parse_args(["--tag", "weather", "--tag", "safety"]).tag
print(tags)
```

```text
['weather', 'safety']
```

`append` preserves order and makes repetition explicit. If no tag is supplied, the selected default controls whether the core sees `[]` or `None`. Choose that contract deliberately.

Counted flags support several detail levels

```python
verbosity_parser = argparse.ArgumentParser(prog="detail-demo")
verbosity_parser.add_argument("-v", "--verbose", action="count", default=0)

for argv in ([], ["-v"], ["-vv"]):
    print(argv, verbosity_parser.parse_args(argv).verbose)
```

```text
[] 0
['-v'] 1
['-vv'] 2
```

An explicit zero default prevents the absent value from being `None`. Later, the logging lesson will translate 0, 1, and 2 into logging levels.

Short options are conveniences, not a second secret vocabulary. Reserve `-v`, `-q`, or `-o` for conventions the audience is likely to recognize, and always provide descriptive long forms for less obvious choices. Argparse accepts `-vv` for a counted `-v` and often accepts `--limit=3` as well as `--limit 3`; show the ordinary spelling in help and examples rather than requiring users to discover every compact form. Never reuse one short letter for different meanings inside related subcommands unless the distinction is genuinely clear.

5. Reject invalid command syntax before the core runs

`choices` is ideal for a short, stable set:

```python
status_parser = argparse.ArgumentParser(prog="status-demo")
status_parser.add_argument(
    "--status",
    choices=("open", "limited", "closed"),
    default="open",
)
```

PYTHON (CONTINUED)

```
print(status_parser.parse_args(["--status", "closed"]).status)
```

TEXT

```
closed
```

An unsupported value produces usage and status 2. It should not reach a core function that silently guesses.

For a value with a compact syntax rule, write a converter:

PYTHON

```
def positive_count(text):
    """Convert text to a positive integer for argparse."""
    try:
        value = int(text)
    except ValueError as error:
        raise argparse.ArgumentTypeError("must be an integer") from error
    if value < 1:
        raise argparse.ArgumentTypeError("must be at least 1")
    return value


count_parser = argparse.ArgumentParser(prog="count-demo")
count_parser.add_argument("--limit", type=positive_count, default=20)
print(count_parser.parse_args(["--limit", "4"]).limit)
```

TEXT

```
4
```

The converter owns **command syntax**: can this token represent a positive count? A domain function still owns questions such as whether 4 reports are available or allowed for this account. That separation keeps the core reusable when values come from a web request or another Python caller.

Capture a usage error to inspect its channel and status:

PYTHON

```
import contextlib
import io

error_output = io.StringIO()

try:
    with contextlib.redirect_stderr(error_output):
        count_parser.parse_args(["--limit", "zero"])
except SystemExit as error:
    usage_status = error.code

print(usage_status)
print("must be an integer" in error_output.getvalue())
```

TEXT

```
2
True
```

Argparse owns this grammar failure. Later processing failures usually use a different application status, often 1, and a message written by the application boundary.

Prefer Path values over an already-open file here

`argparse.FileType` can open files during parsing, but passing a `pathlib.Path` or the special `"-"` string gives the application boundary control over encoding, lifetime, cleanup, and error translation. Unit 9 already established those responsibilities. :::

Checkpoint: choose the clearest argument shape

Choose the clearest argument shape

Choose the argparse design that matches the stated command contract.

1. A user may write –tag river –tag bridge, and order matters. Which action matches that contract?

- **A.** store_true
- **B.** count
- **C.** help
- **D.** append

2. Which design makes –strict false when absent and true when present?

- **A.** action="store_true"
- **B.** type=bool
- **C.** action="append"
- **D.** choices=(False, True)

3. What should the positive_count argparse converter decide?

- **A.** Whether the user’s account permits twenty reports
- **B.** Whether one token represents an integer of at least 1
- **C.** How the final report is rendered
- **D.** Whether the input file exists three stages later

Write your choices before checking the answer key at the end of this book.

6. Give different jobs focused subcommands

One command with fifty flags often represents several jobs forced together. Subcommands give each job its own focused arguments and help:

PYTHON

```
import argparse


def build_parser():
    parser = argparse.ArgumentParser(
        prog="trail-report",
        description="Summarize and validate trail checkpoint reports.",
    )
    subcommands = parser.add_subparsers(dest="command", required=True)

    summarize = subcommands.add_parser(
        "summarize", help="count reports by checkpoint status"
    )
    summarize.add_argument("input", metavar="INPUT")
```

PYTHON (CONTINUED)

```
    summarize.add_argument("--limit", type=positive_count, default=20)
    summarize.set_defaults(handler="summarize")

    check = subcommands.add_parser(
        "check", help="emit reports matching one status"
    )
    check.add_argument("input", metavar="INPUT")
    check.add_argument(
        "--status",
        choices=("open", "limited", "closed"),
        default="closed",
    )
    check.add_argument("--tag", action="append", default=[])
    check.add_argument("--strict", action="store_true")
    check.set_defaults(handler="check")

    return parser
```

Inspect three calls directly:

PYTHON

```
parser = build_parser()

cases = [
    ["summarize", "signals.txt"],
    ["summarize", "signals.txt", "--limit", "5"],
    ["check", "-", "--status", "open", "--tag", "river", "--strict"],
]

for argv in cases:
    print(parser.parse_args(argv))
```

The namespace contains only data needed by the selected subcommand plus the dispatch marker. A later `main()` can choose a callable, but the core should not receive the namespace itself. A namespace couples domain behavior to the CLI framework and makes another caller invent fake command arguments.

`set_defaults(handler="check")` uses a string here so the parsing example stays self-contained. A real application may store a function instead:

PYTHON

```
def run_summary(arguments):
    return f"summarize {arguments.input} with limit {arguments.limit}"


def run_check(arguments):
    return f"check {arguments.input} for {arguments.status}"
```

Then `set_defaults(handler=run_summary)` supports `arguments.handler(arguments)`. Both designs require an explicit adapter that extracts ordinary values before calling reusable application logic.

Do not create subcommands only for decoration

If the program performs one job and merely changes format, one parser with a `--format` option may be clearer. Use subcommands when verbs have meaningfully different inputs or workflows. Ask a new user whether `convert --format json` or `json` as a mysterious subcommand is easier to predict.

7. Improve help by reading it as a new user

Help is generated, but it is still authored. Inspect each level:

```python
parser = build_parser()

root_help = parser.format_help()
print("summarize" in root_help)
print("check" in root_help)
```

Subcommand help is separate:

```python
check_parser = next(
    action
    for action in parser._actions
    if isinstance(action, argparse._SubParsersAction)
).choices["check"]

print(check_parser.format_usage())
```

The `_actions` access above deliberately inspects an internal attribute for a short learning experiment. Production code should normally keep the returned subparser while building, or test help through public process behavior, rather than depend on a private implementation detail.

Review help with these questions:

- Does the description say what result the command produces?
- Does every positional have a useful metavar and explanation?
- Are defaults stated when they affect behavior?
- Are choices meaningful rather than internal abbreviations?
- Does each subcommand help mention its special input and output?
- Is the shortest ordinary invocation actually short?
- Can invalid syntax be repaired by reading the usage line?

Do not overload `epilog` with a manual. Give one or two examples, then link to longer documentation in the package README.

Checkpoint: read help and failure evidence

Read help and failure evidence

Use argparse’s observable behavior, not a guess about implementation.

1. What normally happens after argparse handles –help?

- **A.** It returns an empty Namespace and continues
- **B.** It writes help to stderr and exits with 1
- **C.** It writes help and raises SystemExit with status 0
- **D.** It raises ValueError without output

2. The user supplies –status maybe when choices are open, limited, and closed. Which evidence should the parser produce?

- **A.** A silent default to open
- **B.** A traceback from the domain core
- **C.** A JSON object on stdout
- **D.** Usage and an argparse error on stderr with status 2

3. Why extract ordinary values before calling the reusable core?

- **A.** It prevents the core from depending on argparse’s Namespace and makes other callers natural
- **B.** Namespaces cannot contain strings
- **C.** The shell requires every function to have one argument
- **D.** It makes parser errors use status 0

Write your choices before checking the answer key at the end of this book.

8. Build the trail-report command grammar

Create a parser that satisfies this contract:

1. program name `trail-report` and a useful description;
2. required `summarize` and `check` subcommands;
3. both commands accept positional `INPUT`;
4. `summarize` accepts positive `--limit`, default 20;
5. `check` accepts `--status` from `open`, `limited`, or `closed`, default `closed`;
6. `check` accepts repeatable `--tag`, `--strict`, and counted `-v/--verbose`;
7. both subcommands store a clear dispatch value; and
8. help explains that `-` means stdin.

Start by predicting the namespace for each case:

PYTHON
```
lab_cases = [
    ["summarize", "signals.txt"],
    ["summarize", "signals.txt", "--limit", "5"],
    ["check", "-", "--tag", "river", "--tag", "bridge", "-vv"],
]
```

Then check field values rather than only printing the namespace:

PYTHON
```
lab_parser = build_parser()

summary = lab_parser.parse_args(lab_cases[1])
assert summary.command == "summarize"
assert summary.input == "signals.txt"
assert summary.limit == 5

check = lab_parser.parse_args(lab_cases[2])
assert check.command == "check"
```

PYTHON (CONTINUED)

```
assert check.input == "-"
assert check.tag == ["river", "bridge"]
assert check.verbose == 2
assert check.strict is False
```

Add focused capture checks for `--help`, missing subcommand, invalid status, a non-integer limit, zero limit, and an input named `--draft` after `--`.

Hint 1: sketch the grammar tree

Write `trail-report` at the root. Draw two child verbs. Put shared-looking `INPUT` on both children because argparse subparsers own their arguments. Add only the options meaningful to each child.

Hint 2: implement conversion separately

Write and check `positive_count(text)` before building the parser. It should catch `ValueError`, raise `ArgumentTypeError` with a useful message, reject values below 1, and return the integer otherwise.

Hint 3: capture one exit at a time

Use a fresh `StringIO`, `redirect_stdout` or `redirect_stderr`, and a focused `try/except SystemExit` around one `parse_args()` call. Record both `error.code` and the captured text.

Show one complete parser after attempting the lab

PYTHON

```
import argparse


def positive_count(text):
    """Return a positive integer parsed from text."""
    try:
        value = int(text)
    except ValueError as error:
        raise argparse.ArgumentTypeError("must be an integer") from error
    if value < 1:
        raise argparse.ArgumentTypeError("must be at least 1")
    return value


def build_trail_parser():
    parser = argparse.ArgumentParser(
        prog="trail-report",
        description="Summarize and validate trail checkpoint reports.",
        epilog="Use '-' as INPUT to read from standard input.",
    )
    subcommands = parser.add_subparsers(dest="command", required=True)

    summarize = subcommands.add_parser(
        "summarize", help="count reports by status"
    )
    summarize.add_argument("input", metavar="INPUT")
    summarize.add_argument("--limit", type=positive_count, default=20)
    summarize.set_defaults(handler="summarize")

    check = subcommands.add_parser(
        "check", help="emit reports matching one status"
    )
    check.add_argument("input", metavar="INPUT")
    check.add_argument(
        "--status",
        choices=("open", "limited", "closed"),
        default="closed",
    )
```

PYTHON (CONTINUED)

```
    check.add_argument("--tag", action="append", default=[])
    check.add_argument("--strict", action="store_true")
    check.add_argument("-v", "--verbose", action="count", default=0)
    check.set_defaults(handler="check")
    return parser
```

This solution builds only the grammar. It does not open the input or run the trail logic; those jobs belong to later boundaries.

Key points

- Design example invocations and invalid shapes before writing parser calls.
- The shell turns command text into strings; argparse parses the resulting argument list into converted Python values.
- Positionals identify central subjects, options label values, flags represent switches, repeatable actions collect values, and subcommands separate jobs.
- Use `type`, `choices`, and focused converters for command-syntax validation; keep domain validation in reusable application code.
- Generated help, usage text, and status 2 errors are observable parts of the public command interface.
- Pass an explicit list to `parse_args()` for direct experiments, and do not pass the resulting Namespace into the reusable core.

Continue learning

- Python argparse tutorial
- argparse library reference
- Python `sys.argv` documentation
- Software Carpentry: Command-Line Programs

Put Output, Errors, and Exit Status in the Right Place

Online lesson and runnable notebook

Build a pipe-friendly command that reads standard input, preserves machine-readable stdout, explains anticipated failures on stderr, and reports success through a documented exit status.

- **Level:** Python Foundations
- **Estimated time:** 4–5 hours
- **You will learn:** Route data and diagnostics to separate streams, translate anticipated failures at the command boundary, return meaningful status, and compare direct stream checks with a real child-process check.
- **Practice in:** Google Colab or JupyterLab for `StringIO`, plus a local terminal for pipes, redirects, and exit status

Our parser can describe `trail-report check - --status closed`. The dash says that report lines should arrive through standard input. The next program in a pipeline expects JSON Lines, while a person still needs to know why a damaged line was rejected.

One `print()` for everything cannot satisfy both readers. This lesson gives each result a deliberate channel.

As you work, answer these questions:

- What belongs on stdin, stdout, and stderr?
- Why is stable stdout an interface for another program rather than a screen decoration?
- How is a returned data value different from a process exit status?
- Which exceptions should a command boundary translate into concise messages?
- What does a subprocess check prove that a direct function call cannot?

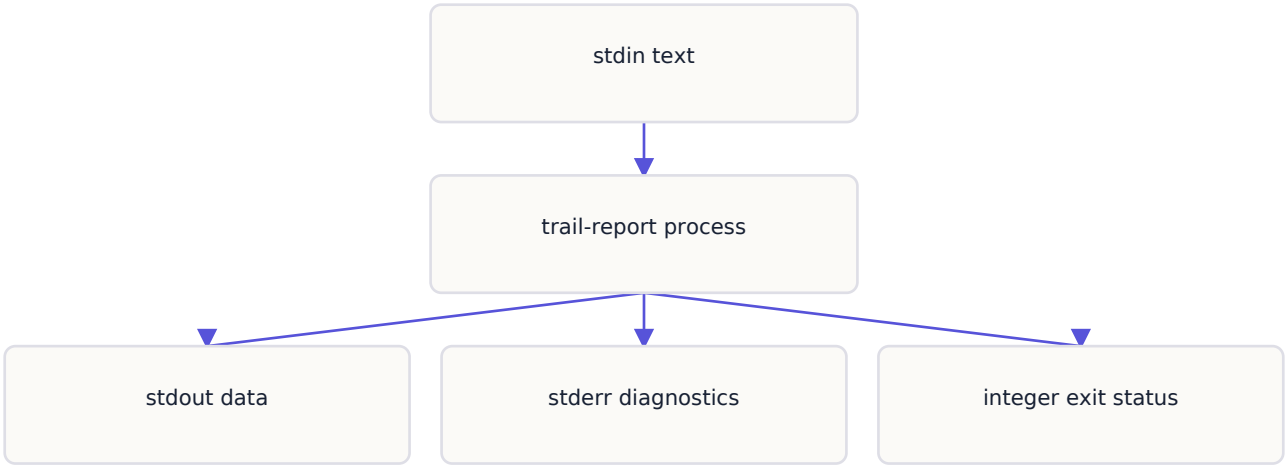
1. Follow one line through a pipeline

A running command normally begins with three text streams:

Stream	Python object	Direction	Typical CLI role
standard input	`sys.stdin`	into the process	records from a pipe, redirected file, or keyboard
standard output	`sys.stdout`	out of the process	the promised result, suitable for another program
standard error	`sys.stderr`	out of the process	diagnostics and operational logs

The process also finishes with an integer **exit status**. That status is not a text stream and not the result data. It is a compact signal to the shell or calling process.

The two outgoing text streams and the exit status are independent observations; a caller can capture each without mixing them.



This shell pipeline sends the first command’s stdout into the second command’s stdin:

```bash
cat signals.txt | trail-report check - | python count_json_lines.py
```

The pipe does not normally carry stderr. A diagnostic can remain visible to a person while clean JSON continues to `count_json_lines.py`.

Common redirections include:

```bash
trail-report check signals.txt > valid.jsonl
trail-report check - < signals.txt
```

Syntax varies between shells, especially for redirecting stderr. The Python design should not depend on the operator using a particular redirect. It should simply honor the stream contract.

2. Read text without assuming a file path

Core record processing can accept any iterable of text lines:

```python
def cleaned_lines(lines):
    """Yield non-blank input lines without their surrounding whitespace."""
    for raw_line in lines:
        line = raw_line.strip()
        if line:
            yield line
```

The function works with a list:

```python
sample = [" north-ridge|open|Windy \n", "\n", " river-gate|closed|Repair \n"]
print(list(cleaned_lines(sample)))
```

```text
['north-ridge|open|Windy', 'river-gate|closed|Repair']
```

It also works when `lines` is an open text file or `sys.stdin`. That is the benefit of accepting the narrow behavior needed—iteration over text—rather than opening a global path inside the core.

At the command boundary, choose the source and own its lifetime:

```python
from contextlib import nullcontext
from pathlib import Path


def open_input(input_name, stdin):
    """Return a context manager for stdin or a UTF-8 text file."""
    if input_name == "-":
        return nullcontext(stdin)
    return Path(input_name).open(encoding="utf-8")
```

`nullcontext(stdin)` lets both branches appear in one `with` statement without closing the process's standard input. The file branch does close the file it opened. This applies the resource-ownership rule from Unit 9.

3. Reserve stdout for the promised data

Suppose each valid trail record becomes a dictionary:

```python
def parse_trail_record(line):
    """Return a validated trail record parsed from one text line."""
    parts = [part.strip() for part in line.split("|", maxsplit=2)]
    if len(parts) != 3:
        raise ValueError("expected CHECKPOINT|STATUS|NOTE")

    checkpoint, status, note = parts
    if not checkpoint:
        raise ValueError("checkpoint cannot be blank")
    if status not in {"open", "limited", "closed"}:
        raise ValueError(f"unknown status {status!r}")
    if not note:
        raise ValueError("note cannot be blank")

    return {"checkpoint": checkpoint, "status": status, "note": note}
```

JSON Lines puts one JSON value on each physical line. It is useful for streaming because a consumer need not wait for one giant array:

```python
import json

record = parse_trail_record("north-ridge|open|Windy")
rendered = json.dumps(record, ensure_ascii=False, sort_keys=True)
print(rendered)
```

```text
{"checkpoint": "north-ridge", "note": "Windy", "status": "open"}
```

The exact key ordering above is part of this teaching example because `sort_keys=True` requests it. A real format contract may care only that keys and values are correct. State which details are stable rather than relying on an accidental representation.

This output is broken for a JSONL consumer:

```text
Processing north-ridge...
{"checkpoint": "north-ridge", "note": "Windy", "status": "open"}
Done!
```

Only the middle line is JSON. Progress announcements belong on stderr or in logging, if they belong at all. Decorative colors, headings, and tables are also unsuitable for a machine-readable stdout mode.

Write to an explicit stream

`print()` accepts a `file` argument:

```python
import io
import json

destination = io.StringIO()
record = {"checkpoint": "north-ridge", "status": "open"}
print(json.dumps(record, sort_keys=True), file=destination)

print(repr(destination.getvalue()))
```

```text
'{"checkpoint": "north-ridge", "status": "open"}\n'
```

That final newline is desirable in JSONL. It lets another line follow without joining two JSON objects.

Decide whether partial output is part of the contract

A streaming command may emit two valid records and then discover a damaged third record. It cannot pull bytes back from a pipe. In this lesson, valid results already written remain on stdout, the diagnostic identifies the later failure, and status 1 says the complete request was not successful.

That policy supports large inputs and visible progress, but it means a caller must inspect status rather than assume every captured line represents a fully successful run. Document it and check it.

Some tasks require all-or-nothing output. They must validate everything before writing, buffer a bounded result, or write to a temporary file and replace the destination only after success. Those designs use more memory or storage and delay the first output. Do not promise atomic output while streaming directly to stdout; the mechanisms contradict the promise.

`--strict` also does not roll back earlier output. It changes what happens **after** the first rejection: stop rather than continue. Output produced before that line remains valid evidence. This distinction will matter in the unit challenge.

Buffering can also affect when a downstream process sees text, even though the logical channel is correct. Newline-terminated `print()` calls are a clear baseline for this course. A long-running interactive pipeline may need an explicit flush policy, but flushing every tiny write has a performance cost. Treat delivery timing as another documented requirement instead of adding `flush=True` everywhere without evidence.

Checkpoint: choose the correct output channel

Choose the correct output channel

Protect the published stream contract in each case.

1. The command promises JSON Lines for another program. Where should a valid record be written?

- **A.** stderr
- **B.** stdout
- **C.** the exit status
- **D.** the parser description

2. Where should a concise explanation of a rejected input line normally go?

- **A.** inside the JSON result on stdout even when the format forbids it
- **B.** into sys.argv
- **C.** stderr
- **D.** into status 0

3. In producer | consumer, which producer channel normally becomes the consumer's stdin?

- **A.** The producer's stdin
- **B.** The producer's status
- **C.** The producer's stderr
- **D.** The producer's stdout

Write your choices before checking the answer key at the end of this book.

4. Send repairable explanations to stderr

An anticipated invalid record should identify its location and reason:

```python
def write_rejection(stderr, line_number, error):
    """Write one stable, concise record diagnostic."""
    print(f"trail-report: line {line_number}: {error}", file=stderr)
```

Check the exact text:

```python
import io

errors = io.StringIO()
write_rejection(errors, 3, "unknown status 'maybe'")
assert errors.getvalue() == "trail-report: line 3: unknown status 'maybe'\n"
```

The message tells a user where to look and what rule failed. It does not include a full traceback because invalid external data is anticipated. A traceback is valuable when debugging an unexpected programming error; it is usually noise when a record merely violates a published input contract.

Do not echo an entire sensitive or enormous record into the diagnostic. A line number, field name, safe identifier, and concise reason are often enough.

5. Return a status instead of returning fake data

This tempting function is ambiguous:

PYTHON
```
def parse_count_bad(text):
    try:
        return int(text)
    except ValueError:
        return 1
```

Does `1` mean “the valid count is one” or “the operation failed”? The caller cannot know. Return data as data and use an exception for an invalid conversion:

PYTHON
```
def parse_count(text):
    """Return an integer count or raise ValueError for invalid text."""
    return int(text)
```

The command boundary can translate that anticipated exception:

PYTHON
```
def count_main(text, stderr):
    """Display no result here; return a command status."""
    try:
        count = parse_count(text)
    except ValueError:
        print(f"count-demo: invalid count {text!r}", file=stderr)
        return 1

    print(f"accepted count {count}")
    return 0
```

The reusable parser has one honest return meaning. The boundary has one status meaning. At the actual process entry point, connect the returned integer to the process:

PYTHON
```
def main():
    return 0


if __name__ == "__main__":
    raise SystemExit(main())
```

`return 1` inside `main()` does not by itself terminate Python with status 1 if someone merely calls and ignores it. `raise SystemExit(main())` is the adapter between the return value and process termination. Keeping it in the entry-point guard also allows another module to call `main()` and inspect its integer.

Use a small documented policy

For this course artifact:

- **0:** every requested record was processed successfully;
- **1:** the command understood the request but an anticipated setting, file, or record failure prevented complete success;

- **2:** argparse rejected the command grammar.

There is no universal requirement that all CLIs use exactly those application codes. There is a strong requirement to document and use one policy consistently. Do not invent a different code for every error sentence unless a real calling program needs those distinctions.

6. Catch only failures the command can explain

This boundary translates known failures:

```python
from pathlib import Path


def read_text_for_command(path, stderr):
    try:
        return Path(path).read_text(encoding="utf-8"), 0
    except FileNotFoundError:
        print(f"trail-report: input not found: {path}", file=stderr)
        return "", 1
    except PermissionError:
        print(f"trail-report: permission denied: {path}", file=stderr)
        return "", 1
    except UnicodeDecodeError as error:
        print(
            f"trail-report: {path}: invalid UTF-8 near byte {error.start}",
            file=stderr,
        )
        return "", 1
```

Each branch can offer a useful next step. An unexpected `AttributeError` caused by a programming mistake is not included. Letting it retain a traceback helps a developer find the defect.

Avoid these broad patterns:

```python
def hides_everything(operation):
    try:
        operation()
    except Exception:
        return 1
```

This valid Python is bad boundary design. It discards the exception type, message, traceback, and user diagnostic. It may turn a programming defect into a silent status that is hard to investigate. Catch narrow anticipated exceptions and preserve context when raising a more useful one.

Do not catch `BaseException` in an ordinary CLI boundary. That also catches termination signals such as `KeyboardInterrupt` and `SystemExit`, interfering with normal process control.

Checkpoint: translate failures at the boundary

Translate failures at the boundary

Keep domain values, exceptions, messages, and process status distinct.

1. Why is returning 1 from parse_count for invalid text a broken contract?

- **A.** One can also be legitimate parsed data, so success and failure are indistinguishable
- **B.** Python forbids functions from returning integers

- **C.** Exit statuses must be strings
- **D.** ValueError can only be raised by argparse

2. What does raise SystemExit(main()) add at the real entry point?

- **A.** It converts stdin to JSON
- **B.** It turns main’s returned integer into the process exit status
- **C.** It catches every exception
- **D.** It prints the integer on stdout

3. Which failure should the file boundary normally translate into a concise user diagnostic?

- **A.** Every programming defect
- **B.** KeyboardInterrupt
- **C.** A known FileNotFoundError for the requested input
- **D.** SystemExit from –help

Write your choices before checking the answer key at the end of this book.

7. Inject streams without binding them too early

Default expressions are evaluated when Python executes the `def` statement. This design captures the current stdout object too early:

```
PYTHON
import sys


def announce_bad(message, stdout=sys.stdout):
    print(message, file=stdout)
```

Explicitly passing a stream still works, but later redirecting or replacing `sys.stdout` may not affect the captured default. Resolve `None` at call time:

```
PYTHON
import sys


def announce(message, stdout=None):
    if stdout is None:
        stdout = sys.stdout
    print(message, file=stdout)
```

Apply the same pattern to stdin, stderr, argument lists, and environment mappings when they must be replaceable:

```
PYTHON
def main(argv=None, stdin=None, stdout=None, stderr=None):
    if stdin is None:
        stdin = sys.stdin
    if stdout is None:
        stdout = sys.stdout
    if stderr is None:
        stderr = sys.stderr
    return 0
```

Do not use `stdin = stdin or sys.stdin`. An empty `StringIO` is a valid supplied stream even if a custom object might be falsey. Test explicitly for `None`.

8. Process records and preserve all three results

Here is a reusable runner for already-open streams:

PYTHON
```
import json


def run_check(lines, stdout, stderr, wanted_status="closed", strict=False):
    """Write matching records and return 0 or 1 after processing lines."""
    had_error = False

    for line_number, raw_line in enumerate(lines, start=1):
        if not raw_line.strip():
            continue

        try:
            record = parse_trail_record(raw_line)
        except ValueError as error:
            write_rejection(stderr, line_number, error)
            had_error = True
            if strict:
                break
            continue

        if record["status"] == wanted_status:
            print(json.dumps(record, ensure_ascii=False, sort_keys=True), file=stdout)

    return 1 if had_error else 0
```

Trace mixed input before running it:

Line	Parse result	Matches `closed`?	stdout	stderr	status so far
1	valid open	no	unchanged	unchanged	0
2	invalid format	n/a	unchanged	line 2 message	1
3	valid `closed`	yes	JSON record	unchanged	1

Now check the exact observations with in-memory streams:

PYTHON
```
import io

input_text = io.StringIO(
    "north-ridge|open|Windy\n"
    "broken line\n"
    "river-gate|closed|Inspection\n"
)
output = io.StringIO()
errors = io.StringIO()

status = run_check(input_text, output, errors)

assert status == 1
assert '"checkpoint": "river-gate"' in output.getvalue()
assert "north-ridge" not in output.getvalue()
```

PYTHON (CONTINUED)

```
assert "line 2" in errors.getvalue()
```

`StringIO` proves the function routes exact text correctly when given these collaborators. It does not prove that module execution, shell argument passing, or the operating system's process status works.

9. Inspect a real child process

`subprocess.run()` can launch a command with an explicit argument list. This self-contained experiment uses Python's `-c` option so no project file is required:

PYTHON

```
import subprocess
import sys

program = (
    "import sys; "
    "text = sys.stdin.read(); "
    "print(text.upper(), end=''); "
    "print('read', len(text), 'characters', file=sys.stderr); "
    "raise SystemExit(0 if text else 1)"
)

completed = subprocess.run(
    [sys.executable, "-c", program],
    input="trail\n",
    text=True,
    capture_output=True,
    check=False,
    timeout=5,
)

assert completed.returncode == 0
assert completed.stdout == "TRAIL\n"
assert completed.stderr == "read 6 characters\n"
```

Important choices:

- `sys.executable` uses the same Python interpreter running the notebook or script;
- the list passes arguments without asking a shell to reinterpret them;
- `input=` supplies the child's stdin;
- `text=True` uses strings rather than bytes;
- `capture_output=True` keeps stdout and stderr separate;
- `check=False` lets us inspect an expected nonzero result without `CalledProcessError`; and
- `timeout=5` prevents a broken experiment from waiting forever.

Avoid `shell=True` for ordinary Python process checks. It adds shell parsing, platform differences, and an injection risk when untrusted text is composed into a command. A shell is useful when the shell itself is the behavior under study; it is unnecessary here.

Checkpoint: inspect direct and process evidence

Inspect direct and process evidence

Decide what each observation actually proves.

1. A StringIO check passes for run_check. What has not yet been proved?

- **A.** The function can write any text
- **B.** JSON is text
- **C.** StringIO has getvalue
- **D.** The installed or module command exposes the same stdout, stderr, and operating-system status

2. Why pass [sys.executable, "-c", program] rather than one shell command string?

- **A.** It avoids unnecessary shell parsing and passes argument boundaries explicitly
- **B.** Lists automatically make every return code zero
- **C.** Python accepts only list input on stdin
- **D.** It merges stderr into stdout

3. Why is check=False useful when verifying a CLI's documented failure case?

- **A.** It disables the child program's validation
- **B.** It returns a CompletedProcess so the expected nonzero status and both streams can be inspected
- **C.** It forces all errors to status 0
- **D.** It skips launching the process

Write your choices before checking the answer key at the end of this book.

10. Build a pipe-friendly trail filter

Extend `run_check()` with this changed requirement:

- ordinary mode continues after invalid records;
- strict mode stops at the first invalid record;
- valid records before that failure remain on stdout;
- either mode returns 1 if a failure occurred; and
- the same invalid record produces the same diagnostic in both modes.

Use this input to make the difference visible:

```python
mixed_reports = (
    "north-ridge|closed|Snow gate\n"
    "broken record\n"
    "river-gate|closed|Bridge review\n"
)
```

Write two `StringIO` checks. Ordinary mode should emit north-ridge and river-gate; strict mode should emit only north-ridge. Both should report line 2 and return 1. Then write a tiny temporary module and repeat one case through `subprocess.run()`.

Hint 1: preserve evidence gathered before failure

Do not erase stdout when an error occurs. Keep a `had_error` boolean, write the diagnostic, and decide only whether to `break` or `continue`.

Hint 2: inject every stream

Let the runner receive `lines`, `stdout`, and `stderr`. Construct fresh `StringIO` objects for each case so ordinary-mode output cannot leak into the strict-mode observation.

Hint 3: assert exact channel boundaries

In addition to finding expected checkpoint names, assert that diagnostics do not appear in stdout and JSON records do not appear in stderr. A correct total with polluted channels is still a broken CLI.

Show the complete runner after attempting the lab

PYTHON

```
import json


def filter_reports(lines, stdout, stderr, wanted_status="closed", strict=False):
    """Write matching JSONL, report invalid records, and return a status."""
    had_error = False

    for line_number, raw_line in enumerate(lines, start=1):
        if not raw_line.strip():
            continue
        try:
            record = parse_trail_record(raw_line)
        except ValueError as error:
            print(f"trail-report: line {line_number}: {error}", file=stderr)
            had_error = True
            if strict:
                break
            continue

        if record["status"] == wanted_status:
            print(json.dumps(record, ensure_ascii=False, sort_keys=True), file=stdout)

    return 1 if had_error else 0
```

The process entry point should call this function with opened input and real or injected output streams, return its integer, and use `raise SystemExit(main())` only under the entry-point guard.

Key points

- Stdin carries input text, stdout carries the promised result, stderr carries diagnostics and logs, and the exit status is a separate integer signal.
- Stable machine-readable stdout makes a CLI composable; progress chatter and diagnostics must not corrupt it.
- Return application status from `main()` and connect it to process termination with `raise SystemExit(main())` at the entry point.
- Keep data and failure meanings separate. Let core conversions raise useful exceptions and translate anticipated boundary failures narrowly.
- Resolve default streams from `None` at call time rather than capturing global stream objects in function defaults.
- `StringIO` checks direct behavior; `subprocess.run()` checks entry-point and operating-system process behavior.

Continue learning

- Python `sys` documentation for standard streams and exit
- Python `io.StringIO` documentation
- Python `subprocess.run` documentation
- Python `contextlib.nullcontext` documentation

Resolve Configuration Without Exposing Secrets

Online lesson and runnable notebook

Merge defaults, TOML, environment variables, and explicit CLI options into one validated settings value while tracing precedence and preventing ordinary secret leaks.

- **Level:** Python Foundations
- **Estimated time:** 4–5 hours
- **You will learn:** Model validated settings, trace a documented precedence order, load TOML and environment strings safely, distinguish omission from explicit values, and redact sensitive input.
- **Practice in:** Google Colab, JupyterLab, or a local Python project with temporary TOML files

The trail-report command now has explicit arguments and clean streams. Operators also want durable preferences: a default report limit, a local TOML file, an environment-specific status, and a one-time command-line override.

If every function reads whichever source it happens to know about, behavior becomes difficult to predict. This lesson resolves raw sources once, validates them, and passes one ordinary settings value into the application.

As you work, answer these questions:

- Why should core code receive one settings value instead of reading global sources repeatedly?
- Which value wins when defaults, a file, the environment, and the CLI disagree?
- How can `None`, `False`, `0`, an empty string, and an absent value mean different things?
- Where should text settings become validated integers or booleans?
- How can a secret leak even when no line deliberately prints it?

1. Replace scattered lookups with one settings value

This function is easy to call but difficult to reason about:

```python
import os


def select_reports_bad(records):
    status = os.environ.get("TRAIL_STATUS", "closed")
    limit = int(os.environ.get("TRAIL_LIMIT", "20"))
    return [record for record in records if record["status"] == status][:limit]
```

The signature hides two inputs. A notebook, process, or test can receive a different answer because global environment state changed. Conversion can fail deep inside otherwise reusable work.

Make the dependency visible with a frozen dataclass:

```python
from dataclasses import dataclass
from pathlib import Path


@dataclass(frozen=True)
class Settings:
    status: str = "closed"
    limit: int = 20
    strict: bool = False
    cache_dir: Path = Path(".trail-cache")
```

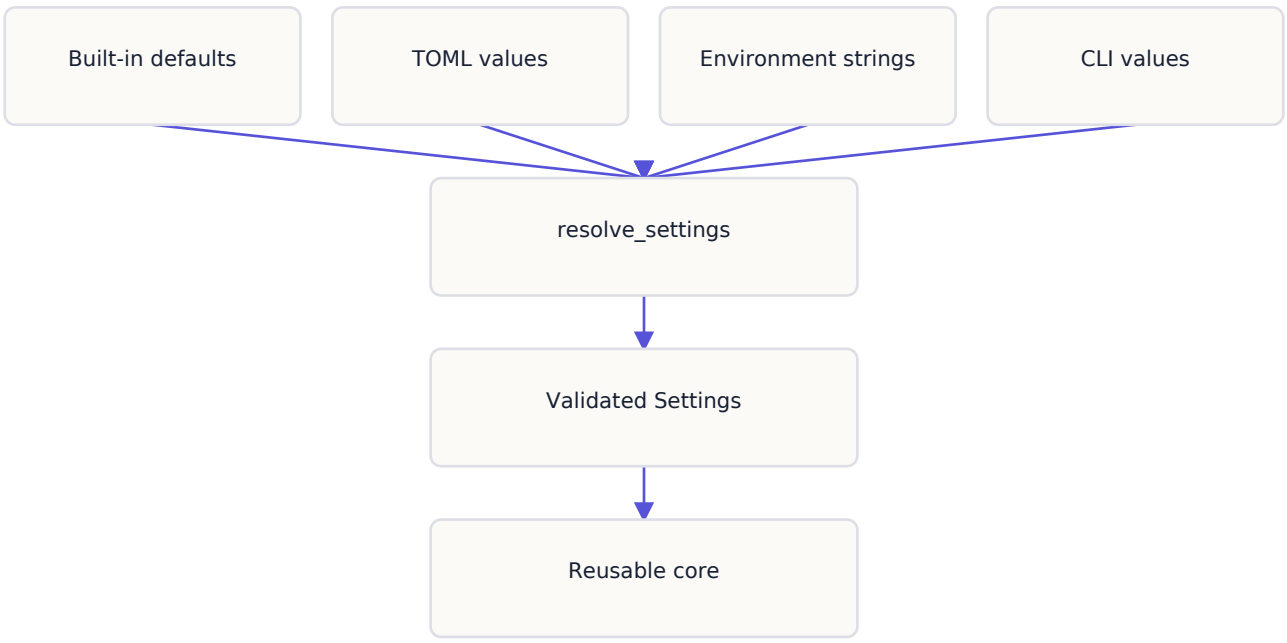
The core accepts the value explicitly:

```python
def select_reports(records, settings):
    selected = [
        record for record in records if record["status"] == settings.status
    ]
    return selected[: settings.limit]
```

One settings object gives the core typed, validated fields. `frozen=True` means the object cannot be reassigned field by field after resolution, reducing surprising mid-run changes. It does not turn referenced mutable objects into immutable objects, so use immutable field values here.

The dataclass is not a universal configuration framework. It is a clear result of a boundary process.

Raw configuration belongs outside the reusable core; one resolver converts and validates it before any records are processed.



2. Write the precedence rule before implementing it

Use this unit’s rule from weakest to strongest:

1 built-in default;

2 TOML file;

3 environment variable; and

4 explicit CLI option.

The strongest supplied source wins for each field. A table makes conflict resolution reviewable:

Setting	Default	TOML	Environment	CLI	Winner
`limit`	20	50	`"30"`	omitted	30 from environment
`status`	closed	limited	absent	open	open from CLI
`strict`	false	true	`"false"`	omitted	false from environment

Notice that `False` can be the winning explicit value. A resolver must not use truthiness to decide whether a source was supplied:

PYTHON
```
def choose_bad(cli_value, fallback):
    return cli_value or fallback


print(choose_bad(False, True))
print(choose_bad(0, 20))
```

TEXT
```
True
20
```

Both answers discard meaningful explicit values. Use `None` as the omission sentinel when `False`, `0`, or `""` can be real input:

PYTHON
```
def choose(cli_value, fallback):
    if cli_value is not None:
        return cli_value
    return fallback


print(choose(False, True))
print(choose(0, 20))
```

TEXT
```
False
0
```

This affects parser design. If a setting can come from weaker sources, an omitted CLI option should often default to `None`, not to the final application default. Otherwise argparse would always appear to provide the strongest value.

Trace one field through ordered updates

One clear strategy starts with defaults and updates only supplied values:

```python
def resolve_limit(default, file_value=None, env_value=None, cli_value=None):
    value = default
    source = "default"

    if file_value is not None:
        value = file_value
        source = "file"
    if env_value is not None:
        value = env_value
        source = "environment"
    if cli_value is not None:
        value = cli_value
        source = "command line"

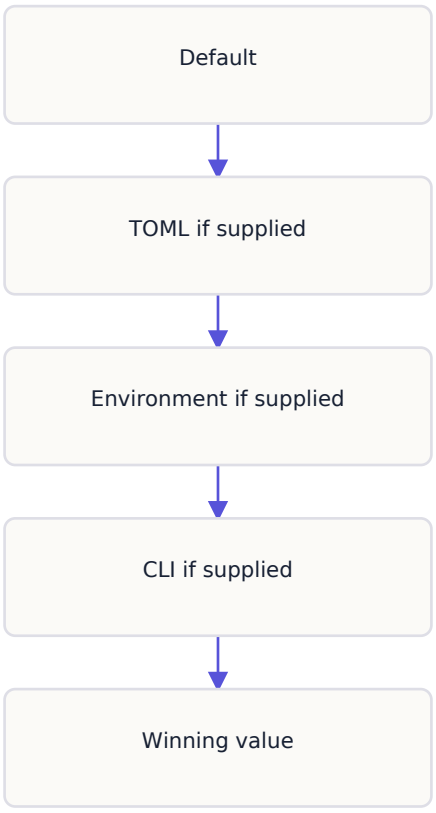
    return value, source


print(resolve_limit(20, file_value=50, env_value=30))
```

```text
(30, 'environment')
```

Tracking a non-sensitive source label helps explain behavior. Do not retain or display raw secrets merely to report provenance.

Read the stack from bottom to top: a stronger supplied source replaces the current value, while an omitted source leaves it unchanged.



Checkpoint: trace the winning setting

Trace the winning setting

Apply default < TOML < environment < explicit CLI for each field.

1. limit is 20 by default, 50 in TOML, 30 in the environment, and omitted on the CLI. What wins?

- **A.** 20
- **B.** 50
- **C.** 30
- **D.** No value

2. TOML sets strict=true, but the strongest supplied source explicitly sets false. What should the resolver produce?

- **A.** true because it is truthy
- **B.** None
- **C.** an argparse error
- **D.** false

3. Why might an argparse option use default=None when weaker configuration sources exist?

- **A.** None records omission so a file or environment value can still win
- **B.** None converts every input to text
- **C.** It makes the option required
- **D.** It disables validation

Write your choices before checking the answer key at the end of this book.

3. Load one real TOML shape

TOML is readable and Python 3.11+ includes the `tomllib` reader. A small file named `trail-report.toml` might contain:

```toml
[trail]
status = "limited"
limit = 50
strict = true
cache_dir = "cache"
```

`tomllib` reads bytes and returns nested dictionaries:

```python
from pathlib import Path
import tomllib


def load_toml(path):
    """Load one TOML document and return its trail table."""
```

PYTHON (CONTINUED)

```
    config_path = Path(path)
    with config_path.open("rb") as source:
        document = tomllib.load(source)

    trail = document.get("trail", {})
    if not isinstance(trail, dict):
        raise ValueError("[trail] must be a table")
    return trail
```

Use a temporary directory for a reproducible example:

PYTHON

```
from tempfile import TemporaryDirectory

with TemporaryDirectory() as directory:
    config_path = Path(directory) / "trail-report.toml"
    config_path.write_text(
        '[trail]\nstatus = "limited"\nlimit = 50\nstrict = true\n',
        encoding="utf-8",
    )
    values = load_toml(config_path)

print(values)
```

TEXT

```
{'status': 'limited', 'limit': 50, 'strict': True}
```

`tomllib` parses TOML types, but it does not know the application's contract. The resolver must reject unknown keys and wrong application-level values:

PYTHON

```
ALLOWED_FILE_KEYS = {"status", "limit", "strict", "cache_dir"}


def reject_unknown_keys(values):
    unknown = sorted(set(values) - ALLOWED_FILE_KEYS)
    if unknown:
        names = ", ".join(unknown)
        raise ValueError(f"unknown [trail] setting(s): {names}")
```

Silently ignoring `limti = 50` would make the user believe a setting applied when it did not. A concise startup failure is safer.

Resolve relative paths from the configuration file

If the file contains `cache_dir = "cache"`, the least surprising meaning is usually a directory beside the configuration file—not a directory beneath whatever current working directory launched the process:

PYTHON

```
def resolve_config_path(value, config_path):
    path = Path(value)
    if path.is_absolute():
        return path
    return Path(config_path).parent / path
```

Normalize only when the application needs it. `Path.resolve()` may touch filesystem assumptions; a simple joined path can preserve intent for later resource handling.

Treat parser failures and policy failures separately

Malformed TOML raises `tomllib.TOMLDecodeError`. A missing optional file may mean “use defaults,” while an explicitly requested missing file should usually be an error. That policy belongs to the command boundary, not `load_toml()`. Keeping the loader focused makes both cases expressible.

4. Convert environment strings deliberately

Environment names and values are strings. Read an injected mapping so the resolver does not depend on the process global:

PYTHON
```
def parse_env_bool(name, text):
    normalized = text.strip().casefold()
    if normalized in {"1", "true", "yes", "on"}:
        return True
    if normalized in {"0", "false", "no", "off"}:
        return False
    raise ValueError(f"{name} must be true/false, yes/no, on/off, or 1/0")
```

Do not use `bool(text)`:

PYTHON
```
print(bool("false"))
print(bool("0"))
```

TEXT
```
True
True
```

Both strings are non-empty. Explicit accepted forms make behavior predictable.

Use one prefix to prevent collisions:

PYTHON
```
def load_environment(environment):
    """Convert supported TRAIL_REPORT_* strings to partial settings."""
    values = {}

    if "TRAIL_REPORT_STATUS" in environment:
        values["status"] = environment["TRAIL_REPORT_STATUS"]
    if "TRAIL_REPORT_LIMIT" in environment:
        raw_limit = environment["TRAIL_REPORT_LIMIT"]
        try:
            values["limit"] = int(raw_limit)
        except ValueError as error:
            raise ValueError("TRAIL_REPORT_LIMIT must be an integer") from error
    if "TRAIL_REPORT_STRICT" in environment:
        values["strict"] = parse_env_bool(
            "TRAIL_REPORT_STRICT", environment["TRAIL_REPORT_STRICT"]
        )

    return values
```

Presence and content are different. An absent `TRAIL_REPORT_STATUS` means no environment override. An empty present value is supplied input and should usually fail validation rather than silently act absent.

PYTHON
```
print(load_environment({}))
print(load_environment({"TRAIL_REPORT_STRICT": "off"}))
```

```text
{}
{'strict': False}
```

Checkpoint: convert and validate raw settings

Convert and validate raw settings

Distinguish source parsing from the application’s final settings contract.

1. Why should TRAIL_REPORT_STRICT="false" use an explicit converter instead of bool(text)?

- **A.** Environment values are already booleans
- **B.** bool("false") is True because the string is non-empty
- **C.** bool can only process integers
- **D.** The environment automatically lowercases values

2. Why reject an unknown TOML key such as limti?

- **A.** TOML tables cannot have keys
- **B.** Every key is a secret
- **C.** Silently ignoring a likely typo makes the user believe configuration was applied
- **D.** Unknown keys always corrupt Python

3. A config file beside data contains cache_dir="cache". Which base is most predictable for that relative path?

- **A.** The Python installation directory
- **B.** The user’s home in every case
- **C.** The notebook server’s startup directory
- **D.** The configuration file’s parent directory

Write your choices before checking the answer key at the end of this book.

5. Validate once after merging sources

Keep source loaders partial, then build the complete object:

```python
VALID_STATUSES = {"open", "limited", "closed"}


def validate_settings(values):
    status = values["status"]
    limit = values["limit"]
    strict = values["strict"]
    cache_dir = Path(values["cache_dir"])

    if status not in VALID_STATUSES:
        raise ValueError(f"status must be one of {sorted(VALID_STATUSES)}")
    if isinstance(limit, bool) or not isinstance(limit, int) or limit < 1:
        raise ValueError("limit must be an integer of at least 1")
    if not isinstance(strict, bool):
```

PYTHON (CONTINUED)
```
        raise ValueError("strict must be true or false")

    return Settings(status, limit, strict, cache_dir)
```

The explicit `isinstance(limit, bool)` check matters because `bool` is a subclass of `int` in Python. `True` should not accidentally become a limit of one.

Merge in precedence order:

PYTHON
```
DEFAULT_VALUES = {
    "status": "closed",
    "limit": 20,
    "strict": False,
    "cache_dir": Path(".trail-cache"),
}


def merge_settings(file_values, env_values, cli_values):
    merged = dict(DEFAULT_VALUES)
    sources = {name: "default" for name in merged}

    for source_name, values in (
        ("file", file_values),
        ("environment", env_values),
        ("command line", cli_values),
    ):
        for name, value in values.items():
            if value is not None:
                merged[name] = value
                sources[name] = source_name

    return validate_settings(merged), sources
```

Check precedence and provenance:

PYTHON
```
settings, sources = merge_settings(
    {"limit": 50, "status": "limited"},
    {"limit": 30},
    {"status": "open", "strict": False},
)

assert settings.limit == 30
assert sources["limit"] == "environment"
assert settings.status == "open"
assert sources["status"] == "command line"
assert settings.strict is False
```

`cli_values` should contain only supported fields extracted from the argparse namespace. Do not use `vars(namespace)` blindly: it may include command, handler, input path, or other adapter details that are not settings.

Explain the result without dumping raw sources

Configuration bugs often sound like “the program ignored my value.” A safe `--show-config` mode can report the final non-sensitive value and its winning source:

PYTHON
```
def describe_settings(settings, sources):
    """Return stable, non-secret lines describing resolved settings."""
```

PYTHON (CONTINUED)

```
    rows = (
        ("status", settings.status),
        ("limit", settings.limit),
        ("strict", settings.strict),
        ("cache_dir", settings.cache_dir),
    )
    return [
        f"{name}={value} (source: {sources[name]})"
        for name, value in rows
    ]
```

PYTHON

```
for line in describe_settings(settings, sources):
    print(line)
```

This report is intentionally assembled from the validated result and safe source labels. Printing the original environment or the complete TOML document would disclose unrelated values and preserve invalid duplicates that did not win. If a future `Settings` object contains a token, omit that field entirely or display only a fixed redaction. Do not rely on remembering to mask one key in a generic dictionary dump.

Also decide which channel owns the report. If the user explicitly requests `--show-config` as the command result, stdout may be correct. If provenance is optional diagnostic detail during a normal JSON run, logging on stderr is correct. “Configuration information” does not determine a channel by itself; the public command contract does.

Distinguish an optional file from an explicitly requested file

Many applications search a conventional path but also accept `--config PATH`. Those cases deserve different missing-file behavior:

- an absent conventional optional file means “continue with weaker defaults”;
- a missing path explicitly supplied by the user is an anticipated error; and
- a present but malformed file is an error in either case because silently ignoring it hides a configuration the user intended to apply.

Keep that decision around the loader. Do not make `load_toml()` catch every `FileNotFoundError` and return `{}` because the loader cannot tell why the path was chosen. The boundary knows whether a default search or an explicit option selected it.

6. Treat environment variables as input, not a vault

An environment variable keeps a secret out of source code, but it does not make the value inherently safe. A secret can leak through:

- shell history when supplied directly as a command argument;
- process inspection on systems where arguments or environments are visible;
- a committed `.env`, TOML, notebook, or example output;
- `print(settings)` or a dataclass representation;
- debug logs and exception context;
- a traceback containing a URL or object representation; or

- copying a real token into a support message or AI prompt.

Avoid this interface:

```bash
trail-report upload --token real-secret-value
```

The value can remain in shell history and process arguments. Prefer a dedicated secret store in production systems. For a small interactive local command, `getpass` can read without echoing:

```python
from getpass import getpass


def request_token():
    token = getpass("Upload token: ")
    if not token:
        raise ValueError("upload token cannot be empty")
    return token
```

`getpass` protects screen echo. It is not encrypted storage, rotation, access control, or a deployment secret manager. Name that limit so learners do not mistake one safe input channel for an entire security system.

Redact representations and diagnostics

Store only what must be used and never include the value in ordinary evidence:

```python
def describe_token(token):
    if token is None:
        return "not configured"
    return f"configured ({len(token)} characters; value redacted)"


print(describe_token("example-not-a-real-token"))
```

```text
configured (24 characters; value redacted)
```

Even length can reveal information in some threat models. It is included here only to show that safe metadata must be an explicit choice. A more cautious program would report only “configured.”

A `.env` file is a convention used by third-party tools, not a protection by itself and not part of Python’s standard library. If a project adopts one, keep real values out of version control, provide a placeholder example, restrict permissions appropriately, and still treat loaded values as untrusted input.

Checkpoint: find the accidental disclosure

Find the accidental disclosure

Choose the behavior that preserves useful evidence without revealing the value.

1. Why is –token real-secret-value a poor default interface?

- **A.** The secret may be exposed in shell history or process arguments
- **B.** Argparse encrypts the value twice

- **C.** Options cannot accept text
- **D.** It always changes stdout to binary

2. What does getpass primarily provide?

- **A.** Permanent encrypted storage
- **B.** An interactive prompt that normally avoids echoing the entered value
- **C.** Automatic token rotation
- **D.** Protection from every traceback and log

3. Which diagnostic is safest for an ordinary run?

- **A.** token=real-value
- **B.** settings={full dataclass repr}
- **C.** upload token configured; value redacted
- **D.** environment={all names and values}

Write your choices before checking the answer key at the end of this book.

7. Resolve trail-report settings from four sources

Build a focused resolver with this contract:

- defaults: status `closed`, limit 20, strict false, cache directory `.trail-cache`;
- optional `[trail]` TOML values;
- `TRAIL_REPORT_STATUS`, `TRAIL_REPORT_LIMIT`, and `TRAIL_REPORT_STRICT`;
- CLI values represented by a mapping whose omitted fields are `None`;
- precedence: default < file < environment < CLI;
- allowed statuses: open, limited, closed;
- limit: integer of at least 1, explicitly rejecting booleans;
- unknown TOML keys fail;
- empty or invalid environment values fail; and
- return `(Settings, sources)` without recording any secret values.

Use temporary files and plain mappings. Check at least these cases:

```python
ordinary_file = {"status": "limited", "limit": 50}
ordinary_environment = {"TRAIL_REPORT_LIMIT": "30"}
ordinary_cli = {"status": "open", "limit": None, "strict": False}
```

The result should have status open from CLI, limit 30 from environment, strict false from CLI, and default cache directory. Add cases for no overrides, file only, an explicit zero, an empty environment string, an unknown file key, and a relative cache path.

Hint 1: normalize one source at a time

Write separate functions for the TOML table and environment mapping. Make each return a partial dictionary whose values are already converted. Do not mix precedence logic into parsing.

Hint 2: update from weakest to strongest

Copy defaults, then loop through `(file, environment, command line)` in that order. Update only values that are not `None`, and update a parallel source dictionary at the same moment.

Hint 3: construct Settings only after validation

Validate the merged status, limit, strict flag, and path once. If validation passes, create the frozen dataclass. The core should never receive a half-valid dictionary.

Show a complete resolver after attempting the lab

PYTHON

```
from dataclasses import dataclass
from pathlib import Path


@dataclass(frozen=True)
class TrailSettings:
    status: str
    limit: int
    strict: bool
    cache_dir: Path


DEFAULTS = {
    "status": "closed",
    "limit": 20,
    "strict": False,
    "cache_dir": Path(".trail-cache"),
}
ALLOWED_KEYS = set(DEFAULTS)
ALLOWED_STATUSES = {"open", "limited", "closed"}


def validate_partial(values, source_name):
    unknown = sorted(set(values) - ALLOWED_KEYS)
    if unknown:
        raise ValueError(f"unknown {source_name} setting(s): {', '.join(unknown)}")
    return values


def environment_values(environment):
    values = {}
    names = {
        "TRAIL_REPORT_STATUS": "status",
        "TRAIL_REPORT_LIMIT": "limit",
        "TRAIL_REPORT_STRICT": "strict",
    }
    for environment_name, setting_name in names.items():
        if environment_name not in environment:
            continue
        raw = environment[environment_name]
        if setting_name == "limit":
            try:
                values[setting_name] = int(raw)
            except ValueError as error:
                raise ValueError(f"{environment_name} must be an integer") from error
        elif setting_name == "strict":
            values[setting_name] = parse_env_bool(environment_name, raw)
        else:
            values[setting_name] = raw
    return values
```

PYTHON (CONTINUED)

```
def resolve_trail_settings(file_values, environment, cli_values):
    partials = (
        ("file", validate_partial(dict(file_values), "file")),
        ("environment", environment_values(environment)),
        ("command line", validate_partial(dict(cli_values), "CLI")),
    )
    merged = dict(DEFAULTS)
    sources = {name: "default" for name in merged}

    for source_name, values in partials:
        for name, value in values.items():
            if value is not None:
                merged[name] = value
                sources[name] = source_name

    status = merged["status"]
    limit = merged["limit"]
    strict = merged["strict"]
    if status not in ALLOWED_STATUSES:
        raise ValueError("status must be open, limited, or closed")
    if isinstance(limit, bool) or not isinstance(limit, int) or limit < 1:
        raise ValueError("limit must be an integer of at least 1")
    if not isinstance(strict, bool):
        raise ValueError("strict must be true or false")

    settings = TrailSettings(
        status=status,
        limit=limit,
        strict=strict,
        cache_dir=Path(merged["cache_dir"]),
    )
    return settings, sources
```

This solution accepts already-loaded file values so file policy remains at the outer boundary. Connect `load_toml()` only after its missing/explicit-file behavior is specified.

Key points

- Resolve raw sources once and pass one validated settings value to reusable code.
- Document precedence before implementation; this unit uses default < TOML < environment < explicit CLI.
- Use `None` for omission when false, zero, or empty values have distinct meanings. Never choose configuration by truthiness alone.
- Read one real format completely, reject unknown keys, convert environment strings explicitly, and resolve relative paths from the configuration file.
- Environment variables are inputs, not automatic secret storage. Protect command history, representations, logs, tracebacks, notebooks, and committed files from sensitive values.
- Track safe source labels when they help explain behavior, but never retain secrets just to show provenance.

Continue learning

- Python `tomllib` documentation
- Python `os.environ` documentation

- Python `getpass` documentation
- Python dataclasses documentation

Write Useful Logs Without Polluting Output

Online lesson and runnable notebook

Separate command results, user diagnostics, and operational log records while configuring handlers once, choosing useful levels, preserving exceptions, and redacting sensitive values.

- **Level:** Python Foundations
- **Estimated time:** 3.5–4.5 hours
- **You will learn:** Distinguish output from diagnostics and logs, create module loggers, configure one application-owned handler, map verbosity to levels, retain exception context, and prevent sensitive values from entering records.
- **Practice in:** Google Colab, JupyterLab, or a fresh local Python interpreter

`trail-report` already sends JSON data to stdout and record-rejection messages to stderr. An operator now asks questions that are different from either result: Which configuration source won? How many records were read? Which file was opened? Where did an unexpected decoder failure occur?

Logging records operational events for later observation. It should make a command easier to operate without changing the data another program receives.

As you work, answer these questions:

- How is a log record different from a result or a concise user diagnostic?
- Why do modules create loggers while the application chooses handlers and formatting?
- Which level fits a routine event, recoverable problem, or failed operation?
- Why do logging calls normally use `%s` placeholders instead of f-strings?
- How can verbose evidence remain useful without disclosing sensitive data?

1. Decide who needs each message

Consider one damaged line. Three messages might be useful:

```text
{"checkpoint": "north-ridge", "status": "closed"}
trail-report: line 4: expected CHECKPOINT|STATUS|NOTE
2026-08-04 14:23:10 WARNING trail_report.runner rejected line=4 reason=field-count
```

They have different audiences and stability promises:

Message	Audience	Mechanism	Stability expectation
JSON record	another program or user requesting data	stdout	documented output schema
concise rejection	user repairing this invocation	stderr write	useful command diagnostic

Message	Audience	Mechanism	Stability expectation
timestamped event	operator or developer investigating behavior	logging	configurable detail and format

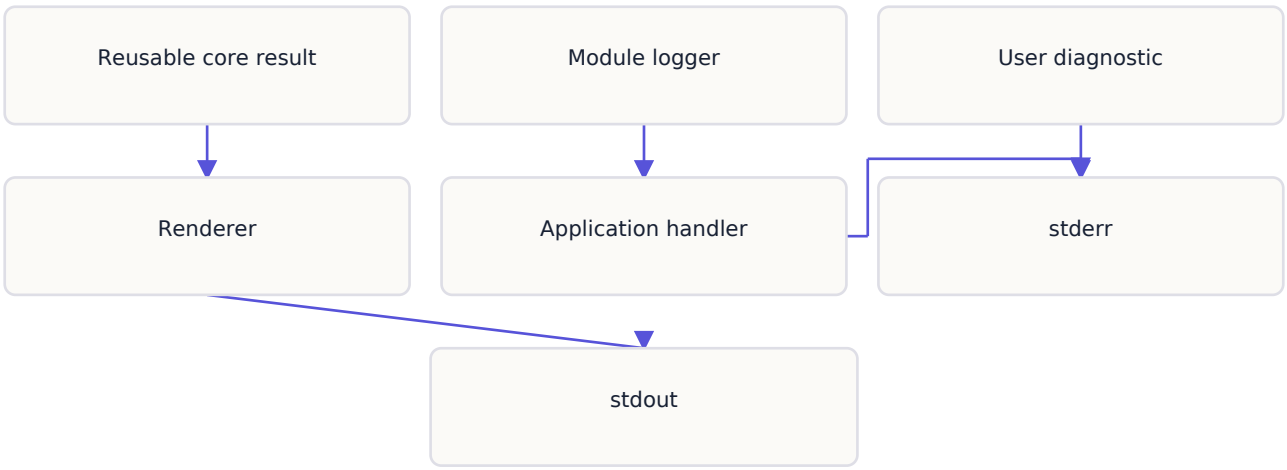
A log record is not automatically better than `print()`. If a user explicitly requested a table, writing that table to stdout is correct. If invalid input needs one immediate repair sentence, writing it to stderr is correct. Logging is valuable when event severity, origin, filtering, and routing matter.

This function pollutes a data stream:

```python
def emit_record_bad(record):
    print("INFO: preparing record")
    print(record)
```

Changing the first call to `logging.info()` helps only if the logging handler is configured for stderr. The command must own both the message choice and the handler destination.

Result data bypasses the logging system; module log records flow through an application-owned handler to stderr.



2. Create a logger for the module that knows the event

Reusable modules conventionally create a logger named after the module:

```python
import logging

logger = logging.getLogger(__name__)
```

If this code is in `trail_report.runner`, the logger name is `trail_report.runner`. The name tells an operator where an event originated and allows configuration for related logger families.

Use it inside work that knows what happened:

```python
def count_statuses(records):
    counts = {"open": 0, "limited": 0, "closed": 0}
    for record in records:
        counts[record["status"]] += 1
    logger.debug("counted %d records", sum(counts.values()))
    return counts
```

The module **emits** a record. It does not decide whether DEBUG records appear, where they go, or how time and severity are formatted. That distinction lets a notebook, CLI, desktop application, and test use the same function with different observation needs.

Do not configure logging during import:

```python
import logging


def configure_during_import_bad():
    logging.basicConfig(level=logging.DEBUG)
```

Calling that function from module top-level would seize a global presentation decision merely because another application imported the code. Libraries should normally create loggers and remain quiet unless the containing application configures handlers.

Logger level and handler level both filter

A record must pass the logger’s effective level and the handler’s level. For a simple CLI, set a clear application logger level and one handler level together rather than creating a mystery of conflicting filters.

```python
def configure_logger(logger, stderr, level):
    handler = logging.StreamHandler(stderr)
    handler.setLevel(level)
    handler.setFormatter(
        logging.Formatter("%(levelname)s %(name)s: %(message)s")
    )

    logger.handlers.clear()
    logger.setLevel(level)
    logger.addHandler(handler)
    logger.propagate = False
    return handler
```

This focused course helper deliberately replaces handlers on the supplied application logger. A larger application may preserve externally installed handlers or configure a hierarchy instead. State ownership before clearing anything.

`propagate = False` prevents the same record from also traveling to an ancestor handler in this self-contained design. It is not a magic setting every library should use; it follows from the application owning this logger’s one handler.

Checkpoint: send each message to the right mechanism

Send each message to the right mechanism

Choose from the public data channel, immediate diagnostic, and configurable operational record.

1. The user selected –format jsonl and another process consumes the output. Which mechanism should emit each record?

- **A.** logger.warning
- **B.** logger.debug
- **C.** stderr
- **D.** the renderer writing to stdout

2. A reusable module wants to report how many records it counted when debugging is enabled. What should it do?

- **A.** Emit a DEBUG record through logging.getLogger(**name**)
- **B.** Call basicConfig during import
- **C.** Print the count into JSON stdout
- **D.** Always open its own log file

3. A user needs to repair malformed line 4 during this run. Which is the clearest baseline behavior?

- **A.** Hide the issue unless DEBUG is enabled
- **B.** Write a concise line-numbered diagnostic to stderr
- **C.** Insert an English sentence into JSONL
- **D.** Return the malformed line as status

Write your choices before checking the answer key at the end of this book.

3. Configure logging once at the application boundary

The CLI knows the selected verbosity and owns stderr. It can configure logging after parsing settings and before running the core:

```python
import logging
import sys


def level_for_verbosity(verbose, quiet=False):
    if quiet:
        return logging.ERROR
    if verbose >= 2:
        return logging.DEBUG
    if verbose == 1:
        return logging.INFO
    return logging.WARNING


application_logger = logging.getLogger("trail_report")
handler = configure_logger(
    application_logger,
    sys.stderr,
    level_for_verbosity(verbose=1),
)
```

The command can now write stable result data independently while modules below `trail_report` emit records. The format can include severity and logger name without requiring every call site to type them.

Why `basicConfig()` can surprise a notebook

`logging.basicConfig()` configures the root logging system only if no relevant handlers already exist, unless `force=True` is deliberately selected. A notebook kernel or imported tool may have configured logging earlier, so changing `basicConfig(level=...)` appears to do nothing.

This experiment makes the state visible:

```python
root = logging.getLogger()
print(len(root.handlers))
print(root.getEffectiveLevel())
```

Do not teach “call `basicConfig()` repeatedly until it works.” In a standalone application, configure once. In a notebook lesson, either use an explicitly owned logger and handler as above or reset only the handlers the lesson created. `force=True` can be a deliberate application-level reset, but a reusable module must not force away another application’s handlers.

Avoid duplicate output

Adding one new handler every time a setup function runs causes duplicates:

```python
def add_handler_bad(logger, stream):
    logger.addHandler(logging.StreamHandler(stream))
```

If called three times, one record may appear three times. Configuration should be idempotent for its declared ownership: reuse, replace, or first remove the handler it owns. The earlier `configure_logger()` clears handlers because its contract says the supplied logger belongs to this small CLI.

Verify configuration with an injected stream

```python
import io

captured_logs = io.StringIO()
demo_logger = logging.getLogger("trail_report.demo")
configure_logger(demo_logger, captured_logs, logging.INFO)

demo_logger.debug("hidden detail")
demo_logger.info("loaded %d records", 3)
demo_logger.warning("one record needs review")

text = captured_logs.getvalue()
assert "hidden detail" not in text
assert "loaded 3 records" in text
assert "one record needs review" in text
```

The direct check proves filtering and routing for this logger. It does not require reading a real log file or relying on global notebook capture.

4. Choose levels from the action they support

Level names are more useful when tied to specific events:

Level	Trail-report event	Why an operator might need it
DEBUG	selected config source or individual parser decision	investigate implementation details
INFO	opened one input and processed 240 records	observe ordinary progress in verbose mode
WARNING	skipped one malformed optional record and continued	notice degraded but continuing work
ERROR	could not read the requested input	understand why the requested operation failed
CRITICAL	application-wide resource makes safe operation impossible	reserve for severe system-level failure

Do not choose ERROR merely because a word sounds dramatic. Ask what happened to the requested operation and what the operator should do.

The default trail-report mode shows WARNING and above. `-v` includes ordinary INFO events; `-vv` includes DEBUG details. `--quiet` raises the threshold to ERROR but does not suppress the explicit result requested on stdout.

Keep messages useful when formatting changes

Call sites should record the event facts, not imitate the final terminal layout. This is useful:

PYTHON
```
logger.info(
    "opened input path=%s encoding=%s",
    input_path,
    "utf-8",
)
```

The application formatter can add level, logger name, time, or another prefix. The call site should not manually prepend `INFO` or a timestamp; doing so would duplicate fields when the formatter changes.

Prefer stable names such as `path=`, `encoding=`, `decoded=`, and `rejected=` when operators search text logs. Do not turn every local variable into a log field. Record only facts that answer a likely operational question, and keep units explicit—`elapsed_ms=42` is clearer than `elapsed=42`.

Logging output is not automatically a forever-stable machine API. If another program must consume events, define and version an explicit structured event format rather than teaching it to scrape human log sentences. The JSONL result contract remains separate from that possible future interface.

Checkpoint: own configuration at the boundary

Own configuration at the boundary

Trace handler ownership and effective filtering.

1. A DEBUG record reaches a logger set to DEBUG, but its only handler is set to INFO. What happens?

- **A.** It is written twice
- **B.** It becomes an ERROR

- **C.** The handler filters it out
- **D.** It moves to stdout automatically

2. Why should a reusable imported module avoid basicConfig()?

- **A.** Modules cannot create log records
- **B.** It prevents all exceptions
- **C.** It changes JSON into bytes
- **D.** It would make a presentation decision that belongs to the containing application

3. A setup function adds a new handler on every call. What symptom is likely after three calls?

- **A.** One event may be emitted three times
- **B.** The parser loses all options
- **C.** stdout closes itself
- **D.** Every level becomes CRITICAL

Write your choices before checking the answer key at the end of this book.

5. Let logging delay message formatting

Prefer logging placeholders:

```python
record_count = 240
logger.info("processed %d records", record_count)
```

The logger receives a format string and arguments. If INFO is filtered out, the logging system need not build the final message. This also gives logging tools a consistent message template.

An f-string is evaluated before the call:

```python
logger.info(f"processed {record_count} records")
```

For a small integer the cost is negligible, but the ownership distinction is important. Avoid calling expensive functions only to build a filtered message:

```python
if logger.isEnabledFor(logging.DEBUG):
    logger.debug("full dependency map: %s", build_dependency_map())
```

Use the explicit guard only when computing the argument itself is expensive. Simple values can be passed directly with placeholders.

6. Preserve exception context in diagnostic mode

Inside an exception handler, `logger.exception()` emits an ERROR record and includes the current traceback:

```python
def load_optional_cache(path, logger):
    try:
        return path.read_text(encoding="utf-8")
    except FileNotFoundError:
        logger.info("cache is absent; starting empty")
        return ""
    except OSError:
        logger.exception("could not read cache path=%s", path)
        raise
```

The missing optional cache has normal fallback behavior and does not need a traceback. Another operating-system failure is re-raised after preserving debugging context. The outer command boundary can decide whether to show a concise user error, enable detailed diagnostics, or let an unexpected failure remain visible.

Do not call `logger.exception()` outside an active exception handler; there is no meaningful current traceback to include. Do not log an exception and then raise it through several layers that all log it again. Choose the boundary that has both useful context and ownership of the diagnostic policy.

7. Redact sensitive and oversized values

Verbose mode is not permission to disclose everything. This call leaks a token:

```python
def log_upload_bad(logger, endpoint, token):
    logger.debug("upload endpoint=%s token=%s", endpoint, token)
```

Repair it by logging a safe fact:

```python
def log_upload(logger, endpoint, token):
    logger.debug(
        "upload endpoint=%s token_configured=%s",
        endpoint,
        token is not None,
    )
```

Also avoid full environment mappings, authorization headers, unbounded record payloads, personal information, and dataclass representations that contain sensitive fields. Prefer counts, safe IDs, source labels, field names, and bounded summaries.

Before adding a record, answer:

1. Who will use this information?
2. At what level should it appear?
3. Could any argument contain a secret or large payload?
4. Will the same failure already be reported elsewhere?
5. Does the message help a concrete investigation?

Checkpoint: read safe diagnostic evidence

Read safe diagnostic evidence

Choose useful records without wasted work, duplicates, or disclosure.

1. Why prefer logger.info("loaded %d records", count) over an f-string for ordinary logging?

- **A.** Percent placeholders change count to a float
- **B.** Logging can delay final message formatting until the record is emitted
- **C.** F-strings always reveal secrets
- **D.** The call is written to stdout instead

2. When is logger.exception most meaningful?

- **A.** Before any operation starts
- **B.** For every successful record
- **C.** Inside an exception handler when the current traceback helps diagnosis
- **D.** Whenever stdout is JSON

3. Which DEBUG record preserves the most useful safe evidence?

- **A.** token=%s with the full token
- **B.** environment=%s with every value
- **C.** record=%s with the entire payload
- **D.** upload endpoint=%s token_configured=%s with no token value

Write your choices before checking the answer key at the end of this book.

8. Add quiet and verbose modes to trail-report

Build an application-owned logging setup with this contract:

1. verbosity 0 emits WARNING and above;
2. verbosity 1 emits INFO and above;
3. verbosity 2 or more emits DEBUG and above;
4. quiet mode emits only ERROR and above;
5. the supplied logger has exactly one handler owned by this setup;
6. the handler writes to the supplied stderr stream;
7. the format is `LEVEL logger.name: message`;
8. processing logs counts and safe checkpoint IDs, never full notes or tokens;
9. JSONL stdout remains identical at every verbosity; and
10. setup called twice does not duplicate records.

Create separate `StringIO` streams for data and logs. Run the same record set at verbosity 0, 1, and 2. Assert exact presence or absence of DEBUG, INFO, WARNING, and ERROR events. Then call setup twice and confirm one warning appears once.

Hint 1: convert verbosity before configuring

Write `level_for_verbosity(verbose, quiet)` as a pure function and check all four outcomes. Do not scatter level arithmetic across handler setup and record processing.

Hint 2: make handler ownership explicit

For this lab, the supplied named logger belongs to the setup function. Clear its existing handlers, attach exactly one configured `StreamHandler`, set its level and formatter, and disable propagation.

Hint 3: compare stdout byte for byte

Run the renderer with the same records and three log levels. Assert `quiet_stdout == info_stdout == debug_stdout`. Log detail may change; public data must not.

Show a complete setup after attempting the lab

PYTHON

```
import logging


def trail_log_level(verbose, quiet=False):
    """Return the application level for CLI verbosity settings."""
    if quiet:
        return logging.ERROR
    if verbose >= 2:
        return logging.DEBUG
    if verbose == 1:
        return logging.INFO
    return logging.WARNING


def configure_trail_logging(logger, stderr, verbose=0, quiet=False):
    """Replace this application's handlers with one stderr handler."""
    level = trail_log_level(verbose, quiet)
    handler = logging.StreamHandler(stderr)
    handler.setLevel(level)
    handler.setFormatter(
        logging.Formatter("%(levelname)s %(name)s: %(message)s")
    )

    logger.handlers.clear()
    logger.setLevel(level)
    logger.addHandler(handler)
    logger.propagate = False
    return logger


def record_run(logger, processed, rejected, token=None):
    """Emit safe operational facts for one completed run."""
    logger.debug("token_configured=%s", token is not None)
    logger.info("processed %d records", processed)
    if rejected:
        logger.warning("rejected %d records", rejected)
```

The CLI boundary should call `configure_trail_logging()` once after resolving arguments and settings. Reusable modules should call their module logger, not this configuration function.

Key points

- Result output, immediate user diagnostics, and operational log records serve different audiences; do not replace all three with one mechanism.
- Reusable modules create named loggers and emit events. The containing application owns handlers, levels, destinations, and formatting.

- Keep a machine-readable stdout clean by routing the CLI’s log handler to stderr.
- Logger and handler levels both filter. Connect `-v`, `-vv`, and `--quiet` to one documented policy.
- Use logging placeholders, and use `logger.exception()` only where active exception context belongs in the diagnostic policy.
- Prevent repeated setup from adding duplicate handlers, and never log secrets, full environments, or unbounded payloads merely because DEBUG is enabled.

Continue learning

- Python Logging HOWTO
- Python logging library reference
- Logging cookbook

Keep the Command Thin and the Core Reusable

Online lesson and runnable notebook

Separate parser, configuration, resource, core, rendering, and process responsibilities; connect one main function to two entry routes; and verify the finished command outside its project.

- **Level:** Python Foundations
- **Estimated time:** 4.5–5.5 hours
- **You will learn:** Untangle a script into explicit layers, keep argparse and global resources outside the core, share one main function across entry points, verify a real process, and choose CLI tools for the problems they actually solve.
- **Practice in:** Google Colab or JupyterLab for direct checks, plus a local package and terminal for installation and process checks

The trail-report behavior now has a command grammar, clean streams, validated settings, and useful logging. It can still become difficult to change if all of those responsibilities live inside one function.

This lesson assembles the pieces as a small application. The command boundary will remain thin: translate process-facing details into ordinary values, call reusable code, render the result, and translate anticipated failures back into diagnostics and status.

As you work, answer these questions:

- Which responsibilities belong in the parser, settings resolver, resource boundary, core, renderer, and `main()`?
- Why should core functions not accept an argparse `Namespace`?
- How can `python -m trail_report` and an installed `trail-report` command call the same implementation?
- What can a direct check prove, and what requires a real child process?
- When would argparse, Click, Typer, or Rich improve the product?

1. Annotate a tangled command before changing it

This function is valid Python and may even work for the first demo:

PYTHON

```
import argparse
import json
import logging
import os
from pathlib import Path


def tangled_main():
    parser = argparse.ArgumentParser()
    parser.add_argument("input")
    parser.add_argument("--status", default=os.environ.get("TRAIL_STATUS", "closed"))
    arguments = parser.parse_args()
    logging.basicConfig(level=logging.INFO)
```

PYTHON (CONTINUED)

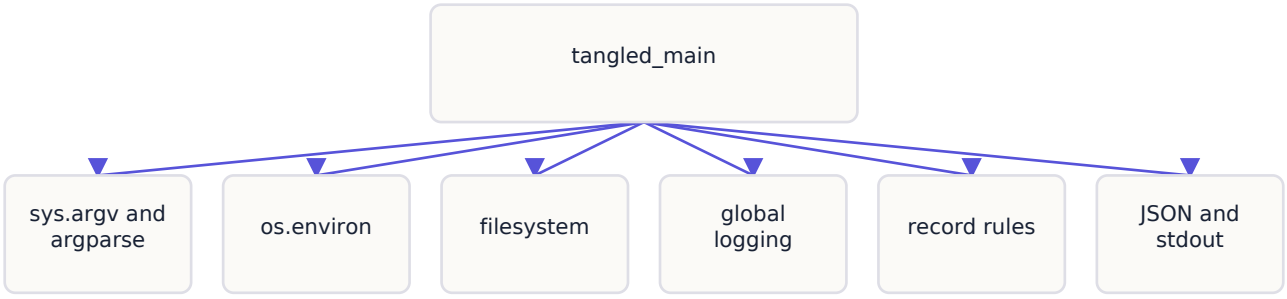
```
    text = Path(arguments.input).read_text(encoding="utf-8")
    for line in text.splitlines():
        checkpoint, status, note = line.split("|", maxsplit=2)
        if status == arguments.status:
            print(json.dumps({"checkpoint": checkpoint, "note": note}))
    return 0
```

Before refactoring, annotate what each line **reads**, **decides**, and **changes**:

Responsibility	Hidden dependency or effect
define and parse command grammar	real `sys.argv`; argparse exits on errors
choose a status	process environment read during parser construction
configure logging	global root handlers
acquire input	filesystem and current directory
parse and select records	mixed with adapters and rendering
render JSON	real stdout
return status	no translation of anticipated failures

The problem is not that any one library is “bad.” The function has too many reasons to change. A new TOML rule, alternate renderer, notebook caller, or file error all require editing the same block.

The tangled function depends directly on every external boundary, so the core behavior cannot be called without constructing process state.



Make one controlled change

Imagine a notebook needs to summarize a list already in memory. It should not invent an input filename, patch `sys.argv`, alter the process environment, or capture global stdout merely to reuse the selection rule. That requirement reveals the first useful extraction: the core should accept records and ordinary settings, then return ordinary result values.

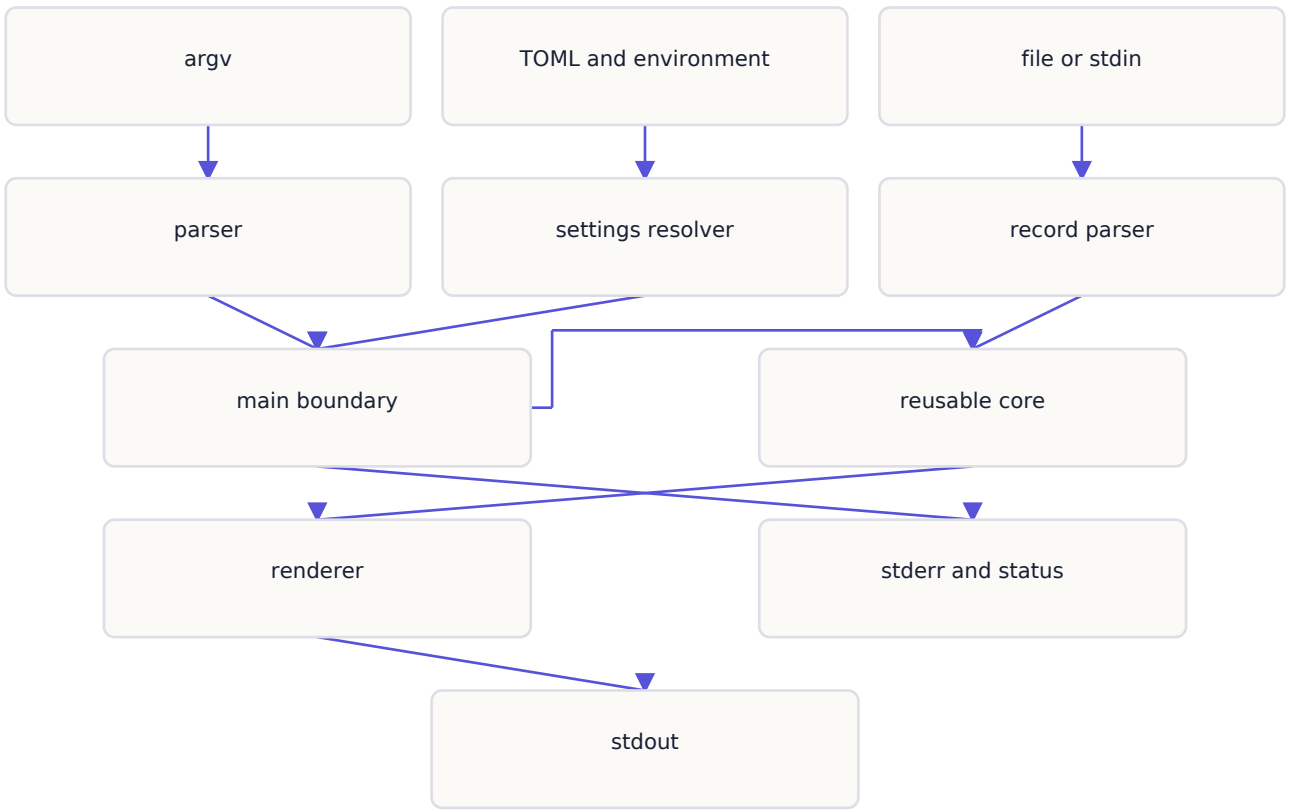
2. Give each boundary one job

A small CLI does not need dozens of layers. It needs enough separation that each function has a clear contract:

1. `build_parser()` describes command syntax and help;
2. `resolve_settings()` converts raw configuration into validated settings;
3. `open_input()` owns files or selects stdin;
4. `parse_records()` turns external lines into domain records;
5. `select_records()` performs reusable application behavior;
6. `render_records()` writes the documented output format;
7. `main()` coordinates adapters, translates anticipated failures, and returns status; and
8. the entry point calls `raise SystemExit(main())`.

The names can vary. The direction of dependency matters more: core functions should not import argparse, read `os.environ`, open global streams, configure logging, or terminate the process.

External details point inward through adapters. The reusable core receives ordinary values and returns ordinary values; it does not point back to argparse or process globals.



This idea is sometimes called a **functional core with an imperative shell**. In ordinary language: keep the decisions as functions over explicit values; keep files, streams, environment state, logging setup, and process termination near the outside. The core does not have to be mathematically pure, but hidden I/O should not be required for an ordinary calculation.

Checkpoint: place each responsibility

Place each responsibility in the right layer

Choose the narrowest owner that preserves reuse and clear evidence.

1. What should select_records normally receive?

- **A.** Ordinary records and validated settings
- **B.** The complete argparse Namespace
- **C.** The process environment mapping and real stdout
- **D.** A request to call sys.exit

2. Which layer should decide whether INPUT="-" means stdin or a UTF-8 file?

- **A.** The status-counting core
- **B.** The resource/command boundary
- **C.** The JSON renderer
- **D.** A dataclass equality method

3. Where should main’s integer become real process termination?

- **A.** Inside every parsing helper
- **B.** Inside the settings dataclass
- **C.** At the entry point with raise SystemExit(main())
- **D.** Inside json.dumps

Write your choices before checking the answer key at the end of this book.

3. Pass domain values instead of a Namespace

An argparse namespace belongs to the CLI adapter:

```python
def select_records_bad(records, arguments):
    return [
        record
        for record in records
        if record["status"] == arguments.status
    ][: arguments.limit]
```

A caller must construct an object with attributes named exactly like CLI destinations. Renaming `--status` to `--state` could break core code even if the domain concept did not change.

Pass ordinary parameters or a settings value:

```python
from dataclasses import dataclass


@dataclass(frozen=True)
class Selection:
    status: str
    limit: int
```

PYTHON (CONTINUED)

```
def select_records(records, selection):
    matching = [
        record
        for record in records
        if record["status"] == selection.status
    ]
    return matching[: selection.limit]
```

The adapter translates names explicitly:

PYTHON

```
def selection_from_arguments(arguments):
    return Selection(status=arguments.status, limit=arguments.limit)
```

That small repetition is valuable. It states which command inputs enter the core and prevents parser-only fields such as `handler`, `command`, or `input` from leaking across the boundary.

Return results before rendering them

When feasible, the core returns records rather than printing them:

PYTHON

```
def summarize_records(records):
    counts = {"open": 0, "limited": 0, "closed": 0}
    for record in records:
        counts[record["status"]] += 1
    return counts
```

One renderer can write JSON and another can write a human table. Changing presentation does not change counting. For very large streams, returning a generator or accepting a result writer may be appropriate, but begin with the smallest interface that meets real size requirements.

4. Assemble one small package

An intentional package tree might be:

TEXT

```
trail-report/
├── pyproject.toml
└── src/
    └── trail_report/
        ├── __init__.py
        ├── __main__.py
        ├── cli.py
        ├── config.py
        ├── records.py
        └── render.py
```

Responsibilities remain easy to state:

- `records.py`: parse, validate, select, and summarize ordinary records;
- `config.py`: load and resolve settings;
- `render.py`: serialize results to a supplied stream;
- `cli.py`: parser, resources, logging setup, `main()`; and

- `__main__.py`: process entry adapter.

A reusable record module

```python
def parse_record(line):
    parts = [part.strip() for part in line.split("|", maxsplit=2)]
    if len(parts) != 3:
        raise ValueError("expected CHECKPOINT|STATUS|NOTE")
    checkpoint, status, note = parts
    if not checkpoint or status not in {"open", "limited", "closed"} or not note:
        raise ValueError("invalid checkpoint, status, or note")
    return {"checkpoint": checkpoint, "status": status, "note": note}


def select_status(records, status, limit):
    return [record for record in records if record["status"] == status][:limit]
```

No CLI import appears. A notebook can call both functions naturally.

A renderer with an explicit destination

```python
import json


def write_jsonl(records, stdout):
    for record in records:
        print(
            json.dumps(record, ensure_ascii=False, sort_keys=True),
            file=stdout,
        )
```

The renderer owns JSON spelling and newlines, but not record selection.

A parser factory

```python
import argparse


def build_parser():
    parser = argparse.ArgumentParser(
        prog="trail-report",
        description="Select trail reports without polluting structured output.",
    )
    parser.add_argument("input", metavar="INPUT")
    parser.add_argument(
        "--status",
        choices=("open", "limited", "closed"),
        default=None,
    )
    parser.add_argument("--limit", type=int, default=None)
    parser.add_argument("-v", "--verbose", action="count", default=0)
    return parser
```

The settings lesson explains why configurable values default to `None`: an omitted CLI option must not erase weaker configuration.

One explicit coordinator

```python
import sys


def main(argv=None, stdin=None, stdout=None, stderr=None, environment=None):
    if stdin is None:
        stdin = sys.stdin
    if stdout is None:
        stdout = sys.stdout
    if stderr is None:
        stderr = sys.stderr
    if environment is None:
        environment = dict()

    arguments = build_parser().parse_args(argv)
    status = arguments.status or environment.get("TRAIL_REPORT_STATUS", "closed")
    limit = arguments.limit if arguments.limit is not None else 20

    try:
        records = [parse_record(line) for line in stdin if line.strip()]
        selected = select_status(records, status, limit)
    except ValueError as error:
        print(f"trail-report: {error}", file=stderr)
        return 1

    write_jsonl(selected, stdout)
    return 0
```

This compact snapshot omits the full TOML and file-input code to keep the architecture visible. Its important boundary is explicit. In the synthesis lab, restore the complete settings resolver and `open_input()` behavior from the earlier lessons.

Do not use `arguments.status or ...` when empty string is meaningful. Here argparse choices prevent an explicit empty status, so the expression is safe; the more general resolver should still test `is not None`.

5. Connect two entry routes to the same `main()`

`__main__.py` supports module execution:

```python
from trail_report.cli import main


if __name__ == "__main__":
    raise SystemExit(main())
```

Now this invocation reaches the same coordinator:

```bash
python -m trail_report --help
```

Unit 10 introduced project scripts. In `pyproject.toml`, an installed console entry point can name the callable:

```toml
[project.scripts]
trail-report = "trail_report.cli:main"
```

An installer creates the platform-appropriate wrapper. The callable returns an integer; the wrapper translates it to process status. Do not create a second implementation for the installed route.

The two routes are valuable evidence:

- `python -m trail_report` checks the package's module entry point; and
- `trail-report` checks the installed project-script metadata and wrapper.

They should expose the same parser, core, output, and status policy.

Checkpoint: choose direct or process evidence

Choose direct or process-level evidence

Match each claim to evidence at the boundary it describes.

1. Which check can prove that the installed trail-report wrapper is discoverable and returns status 1 for bad data?

- **A.** Calling select_records directly
- **B.** Formatting parser help in memory
- **C.** Inspecting the Settings repr
- **D.** Running the installed command with subprocess and inspecting returncode

2. What is the fastest focused evidence for the record-selection rule?

- **A.** Call select_records with ordinary records and assert its returned values
- **B.** Install the package for every example
- **C.** Patch the whole process environment
- **D.** Inspect a shell screenshot

3. Why launch the module or installed command from a directory outside the project?

- **A.** To make JSON sorting faster
- **B.** To expose accidental dependence on the repository's current directory or import path
- **C.** To disable argparse
- **D.** To force every status to 2

Write your choices before checking the answer key at the end of this book.

6. Check each layer before the testing unit

Unit 13 will organize checks with pytest. Here we need enough immediate evidence to know the CLI was assembled correctly.

Check the core directly

```python
records = [
    {"checkpoint": "north-ridge", "status": "open", "note": "Windy"},
    {"checkpoint": "river-gate", "status": "closed", "note": "Repair"},
]

assert select_status(records, "closed", 20) == [records[1]]
assert select_status(records, "open", 0) == []
```

The second assertion documents how slicing treats a zero limit. If the public settings contract rejects zero, test that validation separately rather than pretending the selection algorithm cannot receive it.

Check `main()` with injected collaborators

```python
import io

stdin = io.StringIO("north-ridge|open|Windy\n")
stdout = io.StringIO()
stderr = io.StringIO()

status = main(
    ["-", "--status", "open", "--limit", "5"],
    stdin=stdin,
    stdout=stdout,
    stderr=stderr,
    environment={},
)

assert status == 0
assert '"checkpoint": "north-ridge"' in stdout.getvalue()
assert stderr.getvalue() == ""
```

The compact `main()` snapshot above treats stdin as its source regardless of the input name; the final lab will connect file selection properly. This check still demonstrates explicit arguments and streams.

Check a real process from another directory

After installing the local project into its virtual environment:

```python
import subprocess
from pathlib import Path
from tempfile import TemporaryDirectory

with TemporaryDirectory() as directory:
    completed = subprocess.run(
        ["trail-report", "-", "--status", "open"],
        input="north-ridge|open|Windy\n",
        text=True,
        capture_output=True,
        check=False,
        timeout=10,
        cwd=Path(directory),
    )

assert completed.returncode == 0
assert '"checkpoint": "north-ridge"' in completed.stdout
assert completed.stderr == ""
```

The child cannot rely on the repository as its current directory. If the installed wrapper name is not portable in your environment, build the command list from the environment's scripts directory or verify `[sys.executable, "-m", "trail_report", ...]` first.

Do not turn this one smoke check into the complete testing curriculum. Its role is to prove packaging and process integration before shipping the artifact.

7. Treat command behavior as a public interface

Users and automation may depend on:

- program and option spelling;
- position and meaning of arguments;
- help and usage conventions;
- stdout schema and ordering promises;
- stderr diagnostic prefixes;
- exit-status meanings;
- configuration keys and precedence;
- environment variable names; and
- module and installed entry-point names.

Adding an optional `--limit` with a sensible default can be compatible. Renaming `--status` to `--state`, changing JSON keys, or swapping the meanings of status 1 and 2 is breaking unless a transition is designed.

Not every personal script needs a formal deprecation policy. The moment another person, shell script, scheduled task, or service calls the command, treat observable behavior as an interface. Record intentional changes in release notes, preserve aliases for a stated period when compatibility matters, and never use undocumented output accidents as deliberate guarantees.

Trace a changed requirement through the layers

Suppose a teammate requests `--format text` for people while keeping JSONL as the automation default. A tangled design edits parsing, selection, and printing inside one loop. The layered design makes the impact reviewable:

Layer	Necessary change	Behavior that should not change
parser	add `--format` with `jsonl` and `text` choices	input and status grammar
adapter	pass the selected format to rendering	record selection
renderer	choose one of two presentation functions	configuration precedence
core	none	parsing, validation, filtering, counts
process contract	document both stdout formats	stderr and exit status

Two focused renderers keep branches out of the domain rule:

```python
def write_text(records, stdout):
    for record in records:
        print(
            f"{record['checkpoint']}: {record['status']} — {record['note']}",
            file=stdout,
        )


def render_records(records, output_format, stdout):
    if output_format == "jsonl":
        write_jsonl(records, stdout)
    elif output_format == "text":
        write_text(records, stdout)
    else:
        raise ValueError(f"unsupported output format {output_format!r}")
```

Argparse choices normally prevent the final branch for a CLI caller. Keeping the renderer's own clear contract is still useful because another Python caller can call it directly. The core returns the same selected records in both cases.

Now change the requirement again: text mode should sort by checkpoint, but JSONL must preserve streaming order. Sorting belongs in the text presentation path or in an explicitly named adapter step—not silently inside `select_records()`. Asking "which other behavior should remain unchanged?" is a practical architecture tool, not merely a testing question.

Make startup order visible

Order also carries policy. Parse command syntax before opening a file, so `--help` does not fail because yesterday's input disappeared. Resolve and validate settings before processing the first record, so one invalid environment value cannot produce a partial result. Configure logging before opening external resources if file failures should be recorded. Open the input only after every earlier boundary is ready, and close only resources the application opened.

A clear `main()` reads like that checklist. Clever nesting that opens a file while constructing parser defaults may save one line while making help, cleanup, and partial-output behavior much harder to defend.

8. Choose a tool for the problem it solves

The standard library is enough for this unit. Other tools can improve a real product after requirements justify them:

Tool	Principal value	Good fit	What it does not fix
`argparse`	dependency-free parsing, help, validation, subcommands	standard-library tools and explicit parser design	tangled core, stream contracts, settings policy
Click	composable commands, nesting, decorators, established ecosystem	larger command groups and reusable command components	domain architecture or output discipline by itself
Typer	type-hint-driven command declarations, help, completion	teams that prefer typed function-style command declarations	configuration ownership or secret safety automatically

Tool	Principal value	Good fit	What it does not fix
Rich	styled text, tables, panels, progress, terminal rendering	human-facing presentation modes	parsing, reusable core, machine-readable stdout

A team may combine a parser framework with Rich for a human display and still offer a plain JSON mode. The architecture remains the same: process adapters on the outside, ordinary core values inside.

Evaluate a dependency with concrete questions:

1. Which current requirement becomes simpler or safer?
2. Does the team understand its public interface and release compatibility?
3. What installation, startup, maintenance, and update cost does it add?
4. Can core logic remain independent of framework-specific objects?
5. Is the output still accessible and usable without terminal styling?

Do not rewrite a working argparse command merely because a framework is popular. Do not reject a framework when it clearly reduces repeated complexity. Choose from evidence.

Checkpoint: choose a CLI tool without outsourcing architecture

Choose a CLI tool without outsourcing architecture

Match each requirement to the tool’s primary role.

1. A human-only status view needs readable styled tables, while JSON mode already works. Which tool most directly addresses the new requirement?

- **A.** tomllib
- **B.** subprocess
- **C.** Rich
- **D.** getpass

2. What happens if a tangled script is moved from argparse decorators to another CLI framework without separating its core?

- **A.** All hidden dependencies disappear
- **B.** stdout becomes valid JSON automatically
- **C.** Configuration precedence becomes universal
- **D.** The script remains tangled even if parser declaration becomes nicer

3. Why keep argparse as this unit’s complete baseline?

- **A.** It supplies dependency-free parsing, help, validation, and subcommands while leaving architecture visible
- **B.** It automatically stores production secrets
- **C.** It is the only Python code that can print
- **D.** It replaces package installation

Write your choices before checking the answer key at the end of this book.

9. Ship the complete trail-report package

Assemble the lesson pieces into the package tree shown earlier. Preserve these contracts:

- `build_parser()` accepts `INPUT`, configurable status and limit, strict mode, and verbosity;
- `resolve_settings()` applies default < TOML < environment < CLI;
- `open_input()` maps `-` to supplied stdin and owns only files it opens;
- core record parsing and selection import no argparse, environment, logging configuration, or process-exit functions;
- a JSONL renderer writes only data to supplied stdout;
- concise anticipated failures go to supplied stderr;
- logging is configured once on stderr and does not change JSONL;
- `main()` accepts explicit argv, streams, and environment and returns 0 or 1;
- argparse retains status 2 for grammar errors;
- `__main__.py` and the installed project script call the same `main()`; and
- one subprocess proof runs from outside the repository.

Build evidence in this order:

1. direct core assertions;
2. parser help and invalid-value capture;
3. settings precedence cases;
4. `main()` with `StringIO` and an injected environment;
5. module execution in the project environment;
6. wheel or editable installation into a clean environment; and
7. installed command from an unrelated current directory.

Hint 1: draw imports before writing glue

`records.py` and `render.py` should not import `cli.py`. `cli.py` may import the core, configuration, and renderer. `__main__.py` imports only `main`. If arrows point from core modules back to the command adapter, reconsider the contract.

Hint 2: make main a translation checklist

Inside `main()`: resolve default collaborators, parse arguments, load/resolve settings, configure logging, acquire input, call core, render, translate only anticipated failures, and return status. Keep each step visible rather than nested inside one giant expression.

Hint 3: prove installation rather than source-directory luck

Create a temporary directory and use it as `cwd` for `subprocess.run()`. Pass stdin text, capture both outputs, disable automatic raising for nonzero status, set a timeout, and assert exact public behavior.

Show a compact complete coordinator after attempting the lab

PYTHON

```
import argparse
import io
import json
import os
import sys
from contextlib import nullcontext
from pathlib import Path


def build_complete_parser():
    parser = argparse.ArgumentParser(prog="trail-report")
    parser.add_argument("input", metavar="INPUT")
    parser.add_argument(
        "--status", choices=("open", "limited", "closed"), default=None
    )
    parser.add_argument("--limit", type=int, default=None)
    parser.add_argument("--strict", action="store_true", default=None)
    return parser


def input_context(input_name, stdin):
    if input_name == "-":
        return nullcontext(stdin)
    return Path(input_name).open(encoding="utf-8")


def complete_main(argv=None, stdin=None, stdout=None, stderr=None, environment=None):
    stdin = sys.stdin if stdin is None else stdin
    stdout = sys.stdout if stdout is None else stdout
    stderr = sys.stderr if stderr is None else stderr
    environment = os.environ if environment is None else environment
    arguments = build_complete_parser().parse_args(argv)

    status_name = arguments.status
    if status_name is None:
        status_name = environment.get("TRAIL_REPORT_STATUS", "closed")
    limit = 20 if arguments.limit is None else arguments.limit
    if limit < 1:
        print("trail-report: limit must be at least 1", file=stderr)
        return 1

    try:
        with input_context(arguments.input, stdin) as lines:
            parsed = [parse_record(line) for line in lines if line.strip()]
        selected = select_status(parsed, status_name, limit)
    except (OSError, ValueError) as error:
        print(f"trail-report: {error}", file=stderr)
        return 1

    for record in selected:
        print(json.dumps(record, ensure_ascii=False, sort_keys=True), file=stdout)
    return 0


sample_in = io.StringIO("north-ridge|open|Windy\n")
sample_out = io.StringIO()
sample_err = io.StringIO()
sample_status = complete_main(
    ["-", "--status", "open"],
    stdin=sample_in,
    stdout=sample_out,
    stderr=sample_err,
    environment={},
)
assert sample_status == 0
assert '"north-ridge"' in sample_out.getvalue()
assert sample_err.getvalue() == ""
```

This compact coordinator demonstrates the direction of dependencies. Your package version should reuse the complete settings and logging functions from the preceding lessons instead of copying their behavior into `cli.py`.

Key points

- Untangle a CLI by separating command grammar, configuration, resources, core behavior, rendering, diagnostics, and process termination.
- Keep argparse namespaces and global process resources at the adapter boundary; pass ordinary parameters or validated dataclasses into reusable code.
- Connect module execution and installed project scripts to one callable `main()` implementation.
- Check pure rules directly, check orchestration with injected collaborators, and check packaging and process behavior with a real subprocess from another directory.
- Treat argument spelling, help, output formats, diagnostics, configuration names, and exit statuses as public interfaces once other people or programs depend on them.
- Click, Typer, and Rich can improve specific experiences, but no framework automatically repairs tangled responsibilities or polluted streams.

Continue learning

- Python argparse library reference
- Python `__main__` documentation
- Python subprocess documentation
- Python packaging entry-points specification
- Click documentation
- Typer documentation
- Rich documentation

Unit Challenge: Decode the Deep-Sea Beacon

Online lesson and runnable notebook

Build a streaming command that decodes beacon pings, resolves a shift from explicit sources, preserves clean JSON Lines, reports damaged signals, logs a safe summary, and proves real process behavior.

- **Level:** Python Foundations · Unit 12 challenge
- **Estimated time:** 60–120 minutes
- **Outcome assessed:** Design a discoverable command, preserve its data and diagnostic channels, resolve settings safely, keep the decoder reusable, and verify direct and process behavior.
- **Evidence:** Thirty progressive checks, one debugging record, a real-process proof, and an explanation from a clean rerun

1. Receive a damaged signal from the ocean floor

The research vessel *Asteria* has located two experimental beacons beneath a storm. Each ping contains an identifier and a message encoded with a backward Caesar shift:

```text
NEREID|KHOOR GHSWK
CALYPSO-7|PHHW DW WKH ULGJH
```

Your command, `deepbeacon`, must decode valid pings into JSON Lines so another research tool can consume them. A damaged ping must not corrupt that stream. Instead, the command writes a concise line-numbered explanation to stderr and returns a status that automation can inspect.

The result is satisfying and visible:

```bash
$ deepbeacon pings.txt --shift 3
{"beacon": "NEREID", "message": "HELLO DEPTH"}
{"beacon": "CALYPSO-7", "message": "MEET AT THE RIDGE"}
$ echo $?
0
```

When input contains damage, ordinary mode recovers every valid signal it can:

```bash
$ cat mixed.txt | deepbeacon - --shift 3
{"beacon": "NEREID", "message": "HELLO DEPTH"}
deepbeacon: line 2: expected BEACON|MESSAGE
{"beacon": "CALYPSO-7", "message": "MEET AT THE RIDGE"}
$ echo $?
1
```

The challenge changes the story and record shape from the trail-report lessons. You must select and combine their ideas rather than copy one finished function.

Advance one green group at a time

Implement one responsibility, run only its checks, and record the earliest failure. Open the hints only after you can name the check and evidence blocking your next step. :::

2. Translate the beacon rules into observable behavior

Message decoding

`decode_message(text, shift)` moves ASCII letters **backward** by `shift` places:

- `D` with shift 3 becomes `A`;
- `A` with shift 3 wraps around to `X`;
- lowercase letters remain lowercase;
- spaces, digits, punctuation, and non-ASCII characters remain unchanged;
- shift 0 leaves text unchanged; and
- accepted configured shifts are integers from 0 through 25.

You do not need cryptography knowledge beyond those rules. This is a playful substitution puzzle, not secure encryption.

Ping records

Each non-blank input line must contain exactly one `|`:

```text
BEACON-ID|ENCODED MESSAGE
```

- Surrounding whitespace around both fields is ignored.
- The beacon ID is non-empty and contains only uppercase ASCII letters, digits, and internal hyphens.
- It cannot begin or end with a hyphen.
- The encoded message cannot be blank.
- Blank physical lines are ignored and retain their physical line numbers for later diagnostics.

Configuration and command grammar

The public command is:

```text
deepbeacon [FILE|-] [--shift N] [--strict] [-v]
```

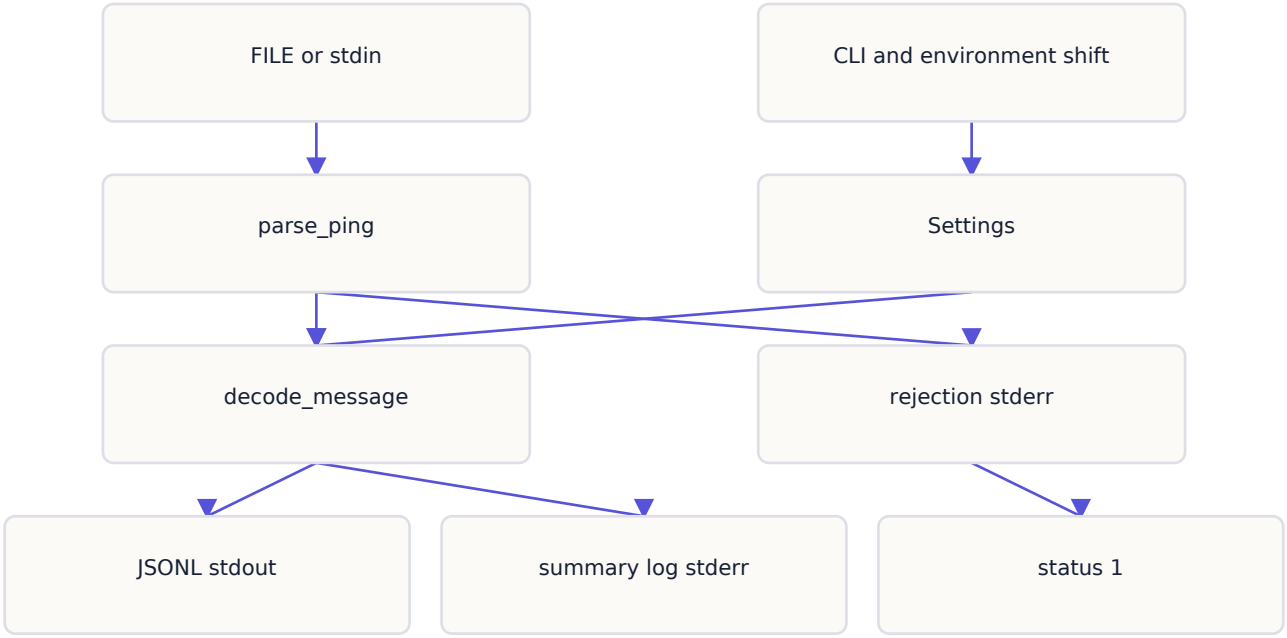
- `FILE` is optional and defaults to `-`, meaning stdin.
- `--shift` has strongest precedence.
- If the CLI value is omitted, `DEEPBEACON_SHIFT` is used when present.
- If both are omitted, the shift is 3.
- Invalid CLI shift syntax is an argparse grammar failure with status 2.
- An invalid environment shift is a configuration failure with status 1.

- `--strict` stops after the first damaged ping; ordinary mode continues.
- `-v` writes one safe `INFO deepbeacon: decoded=N rejected=M` summary to stderr. It never changes stdout.

Channels and status

- Each valid ping becomes one JSON object on stdout with `beacon` and `message` keys.
- Each damaged ping writes `deepbeacon: line N: REASON` to stderr.
- Status is 0 when all processed pings are valid and configuration succeeds.
- Status is 1 after any damaged ping, invalid environment setting, or anticipated file failure.
- Argparse owns status 2 for invalid command syntax.

Notice that decoding is an ordinary function inside the application; file, environment, logging, streams, and process status remain boundary concerns.



3. Start from the contract

Copy this starter into one notebook cell or into `deepbeacon.py`. Required names remove blank-page uncertainty while leaving the central parsing, precedence, channel, and orchestration decisions to you.

PYTHON

```
import argparse
import json
import logging
import os
import sys
from contextlib import nullcontext
from dataclasses import dataclass
from pathlib import Path
```

PYTHON (CONTINUED)

```
@dataclass(frozen=True)
class Settings:
    """Validated behavior for one beacon-decoding run."""

    shift: int
    strict: bool
    verbose: bool


def decode_message(text, shift):
    """Decode ASCII letters by shifting backward; preserve other characters."""
    raise NotImplementedError


def parse_ping(line):
    """Return (beacon_id, encoded_message) from one valid physical line."""
    raise NotImplementedError


def parse_shift(text):
    """Return an integer from 0 through 25 or raise a useful ValueError."""
    raise NotImplementedError


def argparse_shift(text):
    """Adapt parse_shift failures to argparse's command-grammar error."""
    raise NotImplementedError


def resolve_shift(cli_shift, environment):
    """Resolve CLI > DEEPBEACON_SHIFT > default 3."""
    raise NotImplementedError


def build_parser():
    """Build the public deepbeacon command grammar."""
    raise NotImplementedError


def configure_logger(stderr, verbose):
    """Return an application-owned logger whose handler writes to stderr."""
    raise NotImplementedError


def run(lines, settings, stdout, stderr, logger):
    """Decode lines, write both channels, and return processing status."""
    raise NotImplementedError


def input_context(input_name, stdin):
    """Return a context manager for supplied stdin or one opened UTF-8 file."""
    raise NotImplementedError


def main(argv=None, stdin=None, stdout=None, stderr=None, environment=None):
    """Coordinate adapters and return the application status."""
    raise NotImplementedError
```

Do not add Click, Typer, Rich, pytest, or another package. The standard library is sufficient. Do not put the argparse namespace, environment mapping, or real global streams into `decode_message()` or `parse_ping()`.

4. Build one responsibility at a time

Use this order to keep failures local:

1 implement letter shifting and wraparound;

2 parse and validate one ping;

3 validate and resolve the shift;

4 build and inspect the parser;

5 implement `run()` with explicit streams and logger;

6 add ordinary versus strict behavior;

7 select stdin or an owned file;

8 coordinate everything in `main()`; and

9 add the entry-point guard only after direct checks pass.

For letter shifting, `ord()` converts one character to its integer code point and `chr()` converts back. The offset formula can preserve case:

```text
decoded_offset = (encoded_offset - shift) % 26
```

Use base `ord("A")` for uppercase and `ord("a")` for lowercase. Check the character with explicit ASCII ranges; methods such as `.isalpha()` also accept many non-ASCII letters that this challenge promises to preserve.

For `run()`, keep counters named `decoded_count` and `rejected_count`. A boolean or the rejected count can determine final status. Write a rejection immediately so a long stream provides useful progress, but log only one summary after the loop.

5. Run progressive assertions

The helper below lets a plain assertion check a failure without pytest:

```python
def raises_value_error(function, *arguments):
    try:
        function(*arguments)
    except ValueError as error:
        return str(error)
    raise AssertionError("expected ValueError")
```

Run checks 1–9 after the two core functions, 10–17 after shift resolution, and 18–30 after the parser and runner. Do not change expected values to match an incorrect implementation.

```python
import io

# Decoder checks 1–7
assert decode_message("KHOOR", 3) == "HELLO"                        # 1
assert decode_message("khoor", 3) == "hello"                        # 2
assert decode_message("ABC", 3) == "XYZ"                            # 3
assert decode_message("abc", 3) == "xyz"                            # 4
assert decode_message("PHHW 7! 深", 3) == "MEET 7! 深"               # 5
assert decode_message("Stay", 0) == "Stay"                          # 6
assert decode_message("Zz", 25) == "Aa"                             # 7

# Ping checks 8–14
assert parse_ping("NEREID|KHOOR GHSWK") == ("NEREID", "KHOOR GHSWK")  # 8
```

PYTHON (CONTINUED)

```
assert parse_ping("  CALYPSO-7 | PHHW  ") == ("CALYPSO-7", "PHHW")    # 9
assert "expected BEACON|MESSAGE" in raises_value_error(parse_ping, "BROKEN")  # 10
assert "exactly one" in raises_value_error(parse_ping, "A|B|C")       # 11
assert "beacon" in raises_value_error(parse_ping, " |KHOOR")          # 12
assert "message" in raises_value_error(parse_ping, "NEREID| ")        # 13
assert "beacon" in raises_value_error(parse_ping, "bad_id|KHOOR")     # 14

# Shift checks 15–19
assert resolve_shift(None, {}) == 3                                   # 15
assert resolve_shift(None, {"DEEPBEACON_SHIFT": "7"}) == 7           # 16
assert resolve_shift(2, {"DEEPBEACON_SHIFT": "7"}) == 2              # 17
assert resolve_shift(None, {"DEEPBEACON_SHIFT": "0"}) == 0           # 18
assert "0 through 25" in raises_value_error(                           # 19
    resolve_shift, None, {"DEEPBEACON_SHIFT": "30"}
)

# Parser checks 20–21
parser = build_parser()
default_args = parser.parse_args([])
assert (default_args.input, default_args.shift, default_args.strict) == ("-", None, False)  # 20
chosen_args = parser.parse_args(["pings.txt", "--shift", "4", "--strict", "-v"])
assert (chosen_args.input, chosen_args.shift, chosen_args.strict, chosen_args.verbose) == ("pings.txt", ↪
     4, True, True)  # 21

# Runner checks 22–30
quiet_logger = configure_logger(io.StringIO(), verbose=False)
valid_out = io.StringIO()
valid_err = io.StringIO()
valid_status = run(
    ["NEREID|KHOOR GHSWK\n"],
    Settings(shift=3, strict=False, verbose=False),
    valid_out,
    valid_err,
    quiet_logger,
)
assert valid_status == 0                                               # 22
assert valid_out.getvalue() == '{"beacon": "NEREID", "message": "HELLO DEPTH"}\n'  # 23
assert valid_err.getvalue() == ""                                      # 24

mixed_lines = [
    "NEREID|KHOOR\n",
    "damaged\n",
    "CALYPSO-7|PHHW\n",
]
mixed_out = io.StringIO()
mixed_err = io.StringIO()
mixed_status = run(
    mixed_lines,
    Settings(shift=3, strict=False, verbose=False),
    mixed_out,
    mixed_err,
    quiet_logger,
)
assert mixed_status == 1                                               # 25
assert mixed_out.getvalue().count("\n") == 2                           # 26
assert mixed_err.getvalue() == "deepbeacon: line 2: expected BEACON|MESSAGE\n"  # 27

strict_out = io.StringIO()
strict_err = io.StringIO()
run(
    mixed_lines,
    Settings(shift=3, strict=True, verbose=False),
    strict_out,
    strict_err,
    quiet_logger,
)
assert "NEREID" in strict_out.getvalue() and "CALYPSO" not in strict_out.getvalue()  # 28
```

PYTHON (CONTINUED) (CONTINUED)

```
verbose_err = io.StringIO()
verbose_logger = configure_logger(verbose_err, verbose=True)
verbose_out = io.StringIO()
run(
    ["NEREID|KHOOR\n"],
    Settings(shift=3, strict=False, verbose=True),
    verbose_out,
    verbose_err,
    verbose_logger,
)
assert "INFO deepbeacon: decoded=1 rejected=0" in verbose_err.getvalue()  # 29
assert "INFO" not in verbose_out.getvalue()                               # 30
```

After each group passes, restart and run from the top. This exposes hidden notebook state such as an older `Settings` definition or logger handler.

6. Use the hint ladder only when needed

Hint 1

For each character, choose uppercase base, lowercase base, or preservation. Compute `(ord(character) - base - shift) % 26`, add the base again, and call `chr()`. Keep ping parsing separate: split on `|`, verify the part count, strip both fields, then validate beacon and message.

Hint 2

Let `parse_shift()` own the 0–25 rule. `argparse_shift()` should call it and translate `ValueError` to `argparse.ArgumentTypeError`. `resolve_shift()` should return the explicit integer immediately when it is not `None`; otherwise read the environment key if present, then use 3.

Hint 3

In `run()`, enumerate physical lines from 1, skip blanks, catch only `ValueError` from `parse_ping()`, write the rejection, increment a counter, and break only in strict mode. Configure exactly one logger handler on stderr. In `main()`, resolve default collaborators with `is None`, catch configuration and file errors, and return rather than calling `sys.exit()`.

7. Keep debugging evidence

Record one real failure before repairing it:

Field	Your evidence
Smallest failing check	
Expected value, channel, or status	
Actual value, channel, or status	
First line/function that can explain the difference	
One hypothesis	
One controlled change	
Focused rerun result	

Field	Your evidence
Clean checks 1–30 result	

Useful examples include `A` failing to wrap to `X`, a diagnostic appearing on stdout, an environment value overriding an explicit CLI shift, strict mode decoding the third line, or one verbose summary appearing twice. Fix the earliest false assumption instead of editing every downstream symptom.

8. Compare a complete beacon command

Open this only after attempting all progressive groups. Compare contracts and boundary ownership rather than requiring identical local variable names.

Show the complete runnable module

PYTHON

```
import argparse
import json
import logging
import os
import sys
from contextlib import nullcontext
from dataclasses import dataclass
from pathlib import Path


@dataclass(frozen=True)
class Settings:
    """Validated behavior for one beacon-decoding run."""

    shift: int
    strict: bool
    verbose: bool


def decode_message(text, shift):
    """Decode ASCII letters by shifting backward; preserve other characters."""
    decoded = []
    for character in text:
        if "A" <= character <= "Z":
            base = ord("A")
            offset = (ord(character) - base - shift) % 26
            decoded.append(chr(base + offset))
        elif "a" <= character <= "z":
            base = ord("a")
            offset = (ord(character) - base - shift) % 26
            decoded.append(chr(base + offset))
        else:
            decoded.append(character)
    return "".join(decoded)


def parse_ping(line):
    """Return (beacon_id, encoded_message) from one valid physical line."""
    if "|" not in line:
        raise ValueError("expected BEACON|MESSAGE")
    if line.count("|") != 1:
        raise ValueError("expected exactly one '|' separator")

    beacon, message = (part.strip() for part in line.split("|"))
    if not beacon:
        raise ValueError("beacon cannot be blank")
    valid_characters = all(
        character.isascii()
        and (character.isupper() or character.isdigit() or character == "-")
```

PYTHON (CONTINUED)

```
        for character in beacon
    )
    if not valid_characters or beacon.startswith("-") or beacon.endswith("-"):
        raise ValueError("beacon must use uppercase ASCII letters, digits, and internal hyphens")
    if not message:
        raise ValueError("message cannot be blank")
    return beacon, message


def parse_shift(text):
    """Return an integer from 0 through 25 or raise a useful ValueError."""
    try:
        shift = int(text)
    except (TypeError, ValueError) as error:
        raise ValueError("shift must be an integer from 0 through 25") from error
    if not 0 <= shift <= 25:
        raise ValueError("shift must be an integer from 0 through 25")
    return shift


def argparse_shift(text):
    """Adapt parse_shift failures to argparse's command-grammar error."""
    try:
        return parse_shift(text)
    except ValueError as error:
        raise argparse.ArgumentTypeError(str(error)) from error


def resolve_shift(cli_shift, environment):
    """Resolve CLI > DEEPBEACON_SHIFT > default 3."""
    if cli_shift is not None:
        return cli_shift
    if "DEEPBEACON_SHIFT" in environment:
        return parse_shift(environment["DEEPBEACON_SHIFT"])
    return 3


def build_parser():
    """Build the public deepbeacon command grammar."""
    parser = argparse.ArgumentParser(
        prog="deepbeacon",
        description="Decode streaming deep-sea beacon pings as JSON Lines.",
        epilog="Use '-' or omit FILE to read pings from standard input.",
    )
    parser.add_argument("input", nargs="?", default="-", metavar="FILE")
    parser.add_argument(
        "--shift",
        type=argparse_shift,
        default=None,
        metavar="N",
        help="backward shift from 0 through 25",
    )
    parser.add_argument(
        "--strict",
        action="store_true",
        help="stop after the first damaged ping",
    )
    parser.add_argument(
        "-v",
        "--verbose",
        action="store_true",
        help="write one decoding summary to stderr",
    )
    return parser


def configure_logger(stderr, verbose):
    """Return an application-owned logger whose handler writes to stderr."""
```

PYTHON (CONTINUED) (CONTINUED)

```
    logger = logging.getLogger("deepbeacon")
    logger.handlers.clear()
    logger.propagate = False
    logger.setLevel(logging.INFO if verbose else logging.WARNING)

    handler = logging.StreamHandler(stderr)
    handler.setLevel(logging.INFO if verbose else logging.WARNING)
    handler.setFormatter(logging.Formatter("%(levelname)s %(name)s: %(message)s"))
    logger.addHandler(handler)
    return logger


def run(lines, settings, stdout, stderr, logger):
    """Decode lines, write both channels, and return processing status."""
    decoded_count = 0
    rejected_count = 0

    for line_number, raw_line in enumerate(lines, start=1):
        if not raw_line.strip():
            continue
        try:
            beacon, encoded_message = parse_ping(raw_line)
        except ValueError as error:
            print(f"deepbeacon: line {line_number}: {error}", file=stderr)
            rejected_count += 1
            if settings.strict:
                break
            continue

        result = {
            "beacon": beacon,
            "message": decode_message(encoded_message, settings.shift),
        }
        print(json.dumps(result, ensure_ascii=False, sort_keys=True), file=stdout)
        decoded_count += 1

    logger.info("decoded=%d rejected=%d", decoded_count, rejected_count)
    return 1 if rejected_count else 0


def input_context(input_name, stdin):
    """Return a context manager for supplied stdin or one opened UTF-8 file."""
    if input_name == "-":
        return nullcontext(stdin)
    return Path(input_name).open(encoding="utf-8")


def main(argv=None, stdin=None, stdout=None, stderr=None, environment=None):
    """Coordinate adapters and return the application status."""
    stdin = sys.stdin if stdin is None else stdin
    stdout = sys.stdout if stdout is None else stdout
    stderr = sys.stderr if stderr is None else stderr
    environment = os.environ if environment is None else environment

    arguments = build_parser().parse_args(argv)
    try:
        shift = resolve_shift(arguments.shift, environment)
    except ValueError as error:
        print(f"deepbeacon: {error}", file=stderr)
        return 1

    settings = Settings(shift, arguments.strict, arguments.verbose)
    logger = configure_logger(stderr, settings.verbose)
    try:
        with input_context(arguments.input, stdin) as lines:
            return run(lines, settings, stdout, stderr, logger)
    except (OSError, UnicodeError) as error:
        print(f"deepbeacon: cannot read {arguments.input!r}: {error}", file=stderr)
```

PYTHON (CONTINUED) (CONTINUED) (CONTINUED)

```
        return 1


if __name__ == "__main__":
    raise SystemExit(main())
```

The core decoder and ping parser do not know about argparse, global streams, or process termination. `main()` resolves those adapters and translates only anticipated configuration and input failures.

9. Prove the changed requirement and real process

Strict mode is the boundary case that prevents a happy-path-only solution. Use input with valid, invalid, then valid pings. Ordinary mode must decode both valid pings; strict mode must stop before the third line. Both return 1 and both write the same line-2 diagnostic.

After saving the complete implementation as `deepbeacon.py`, run a real child process from a temporary directory:

PYTHON

```
import subprocess
import sys
from pathlib import Path
from tempfile import TemporaryDirectory

script = Path("deepbeacon.py").resolve()
process_input = (
    "NEREID|KHOOR GHSWK\n"
    "damaged\n"
    "CALYPSO-7|PHHW\n"
)

with TemporaryDirectory() as directory:
    completed = subprocess.run(
        [sys.executable, str(script), "-", "--shift", "3", "--strict"],
        input=process_input,
        text=True,
        capture_output=True,
        check=False,
        timeout=10,
        cwd=directory,
    )

assert completed.returncode == 1
assert '"beacon": "NEREID"' in completed.stdout
assert "CALYPSO-7" not in completed.stdout
assert completed.stderr == "deepbeacon: line 2: expected BEACON|MESSAGE\n"
```

Using an absolute script path while changing `cwd` proves that file and stream behavior do not accidentally rely on the source directory. If you package the command, replace the script invocation with the installed `deepbeacon` entry point and keep the unrelated working directory.

Finally run help and invalid CLI shift as processes. Help should use status 0 and stdout. `--shift 30` should use status 2 and argparse’s stderr. Invalid `DEEPBEACON_SHIFT=30` without `--shift` should use status 1 and the application’s concise stderr. These similar-looking failures occur at different boundaries.

10. Check your understanding

Deep-Sea Beacon command check

Connect every answer to stdout, stderr, status, precedence, or process evidence from your artifact.

1. Why must the verbose summary stay off stdout?

- **A.** Logging cannot contain numbers
- **B.** A non-JSON summary would corrupt the JSON Lines contract consumed by another program
- **C.** stderr automatically encrypts it
- **D.** Exit status can contain the whole summary

2. The environment sets shift 7 and the user supplies –shift 2. Which value reaches Settings?

- **A.** 3
- **B.** 7
- **C.** 2
- **D.** 9

3. Why does run return 0 or 1 instead of calling sys.exit?

- **A.** Python processes cannot exit
- **B.** Argparse forbids integers
- **C.** JSON requires a returned status
- **D.** The function remains directly callable; only the outer entry point translates status into process termination

4. What changes when –strict is supplied for valid, damaged, valid input?

- **A.** Processing stops after the damaged line, so the final valid ping is not decoded
- **B.** The damaged line moves to stdout
- **C.** The shift always becomes zero
- **D.** The status becomes 0

5. What does the unrelated-directory subprocess check add beyond calling run with StringIO?

- **A.** It proves the cipher is secure
- **B.** It exercises real entry-point wiring, operating-system streams, working-directory assumptions, and process status
- **C.** It removes the need for direct checks
- **D.** It makes invalid records valid

Write your choices before checking the answer key at the end of this book.

11. Record the challenge result

Evidence rubric

Evidence	Ready to record when
Decoder	ASCII wraparound, case, shift boundaries, and preserved characters pass unchanged.
Parser and configuration	Ping rules, CLI grammar, precedence, and invalid settings produce the documented result.
Channels	JSONL, diagnostics, and the optional log summary stay in their exact channels.
Status	Success, processing/configuration failure, and argparse usage failure remain distinct.
Architecture	Core functions accept ordinary values; `main()` owns adapters and returns status.
Changed requirement	Ordinary and strict mode differ only where the contract says they should.
Reproducibility	Checks 1–30 and the unrelated-directory subprocess proof pass after a clean restart.
Debugging	One evidence record connects a failing observation to one controlled repair.

This button stores a self-reported marker only in this browser. It does not submit work, grade the artifact, verify identity, or issue a certificate.

Key points

- A useful CLI challenge integrates grammar, configuration, resources, reusable behavior, rendering, diagnostics, logging, and process status without mixing them into one function.
- Caesar shifting is the playful domain rule; it remains independently callable and preserves characters outside explicit ASCII ranges.
- Stdout contains only JSON Lines. Rejections and the optional operational summary remain on stderr.
- CLI shift overrides environment shift, which overrides the built-in default; invalid values fail at the boundary that received them.
- Ordinary mode recovers later pings, strict mode stops, and both retain output produced before the rejection.
- Direct assertions explain local behavior; a real subprocess proves the outer entry point, streams, working-directory independence, and process status.

Continue learning

- Python argparse documentation

- Python JSON Lines building blocks
- Python logging documentation
- Python subprocess documentation

Unit 13: Testing Python Programs

Online lesson and runnable notebook

Learn how Unit 13 turns behavioral contracts into focused pytest evidence through readable failures, parametrized boundaries, isolated fixtures, controlled dependencies, coverage questions, regressions, and generated examples.

- **Level:** Python Foundations
- **Estimated time:** 45–60 minutes
- **You will learn:** Prepare a real pytest workspace, distinguish a passing check from proof, and plan the evidence produced by seven connected testing lessons.
- **Practice in:** A local project and terminal for pytest commands, plus Google Colab or JupyterLab for direct function experiments

A weather station should classify a 70-kilometre-per-hour wind as a high-risk event. A test expected `"high"`, but the program returned `"medium"`:

```text
$ pytest -q tests/test_risk.py::test_seventy_is_high
F                                                                    [100%]
================================= FAILURES =================================
_____________________________ test_seventy_is_high _____________________________

    def test_seventy_is_high():
>       assert risk_level(70) == "high"
E       AssertionError: assert 'medium' == 'high'
E
E         - high
E         + medium

1 failed in 0.05s
```

The red `F` is useful, but it does not decide who is wrong. The implementation may mishandle the boundary. The expected value may contradict the written rule. The test may be importing a different installation. Testing begins when you connect the report to a contract and ask which explanation fits the evidence.

Suppose the requirement says:

Wind below 40 is low risk, wind from 40 through 69 is medium risk, and wind of 70 or more is high risk.

Now the failure identifies a violated boundary. A correction followed by a green rerun gives evidence that this case behaves as specified in that environment. It does **not** prove that every wind value, parser, file, clock, or command route is correct. A dependable suite is a collection of deliberately chosen claims, not a spell that removes all defects.

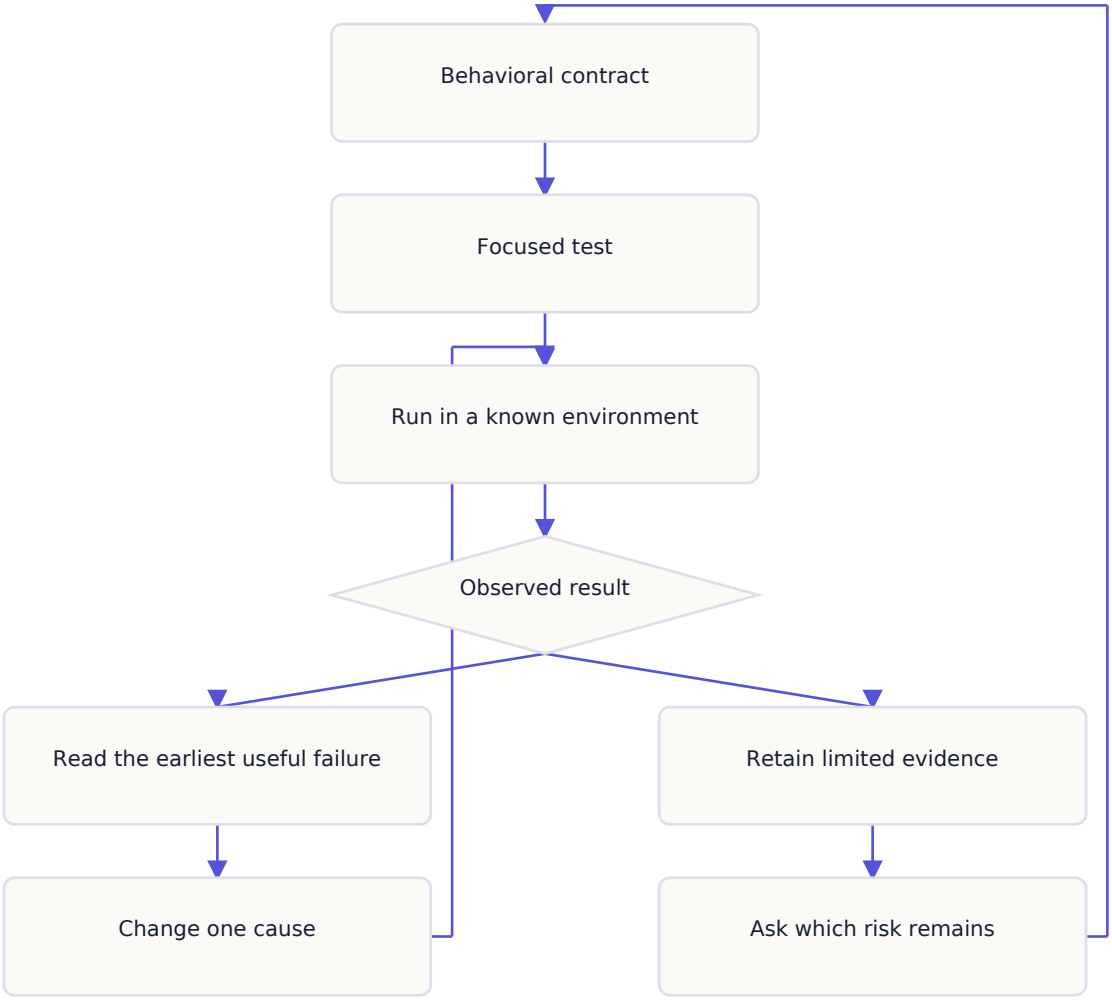
What you will be able to protect

Across this unit you will build tests around a small **Meteor Watch** package. It classifies weather observations, writes alert reports, reads configuration, uses a clock and notification sender, and exposes a thin command boundary. The domain remains simple so that the testing decision is always visible.

By the end, you will be able to:

- turn a written contract into ordinary, boundary, invalid, and regression examples;
- name tests so a failed node explains the scenario and expected result;
- run all tests or one selected file, node, keyword, or previous failure;
- distinguish collection, setup, call, and teardown evidence;
- read pytest’s assertion explanations, value diffs, captured output, and short summary;
- convert a case table into readable parametrized nodes;
- use fixtures for fresh state, temporary paths, output capture, and reliable teardown;
- control time, randomness, environment settings, and collaborators without making tests depend on real services;
- choose the smallest test boundary capable of exposing a stated risk;
- interpret statement and branch coverage as prompts for investigation;
- preserve a discovered bug as a regression test;
- complete a red–green–refactor cycle; and
- use Hypothesis to search a general rule and shrink a failure to a small counterexample.

Notice that a test is part of a feedback loop. A green result supports only the claims that the selected tests actually exercised.



Observed result → Read the earliest useful failure: red

Observed result → Retain limited evidence: green

The seven-stage Meteor Watch build

Every lesson adds a different kind of evidence. Each page includes a complete small snapshot, so a missed fragment does not leave the next lesson with an undefined function.

Step	Lesson	What you practise	Evidence you keep
1	Write Tests That Protect Behavior	Translate a contract into independent public-behavior checks without copying implementation logic.	A behavior table, red/green pair, and explanation of what remains unproved.
2	Run Pytest and Read the Failure	Build a `src` project, collect tests, select nodes, and diagnose collection, setup, and assertion evidence.	An annotated report and one-failure-at-a-time repair record.

Step	Lesson	What you practise	Evidence you keep
3	Turn Examples into Clear Parametrized Cases	Partition inputs, test thresholds, and give every case a useful node ID.	A boundary table and a readable parametrized suite.
4	Build an Isolated Test World with Fixtures	Supply fresh state, manage teardown, use temporary paths, and capture output or logs.	Setup/call/teardown evidence and a state-leak repair.
5	Control Dependencies Without Fragile Mocks	Inject a clock and sender, patch process state, and use a spec-constrained mock only where interaction is public behavior.	Deterministic results and proof that temporary changes are restored.
6	Test at the Smallest Boundary That Can Fail	Match risks to function, integration, and process checks; inspect line and branch coverage.	A risk portfolio, a subprocess result, and one justified coverage addition.
7	Preserve Bugs and Explore General Rules	Preserve a failure, work red–green–refactor, and let Hypothesis shrink a counterexample.	A named regression and minimal generated failure.

The unit challenge changes the story and the code. You will **Clear the Buggy Spaceport for Launch** by writing a focused suite against a published contract, exposing seeded defects, fixing them one at a time, and leaving a clean regression net.

Bring forward the boundaries you already built

Testing combines earlier Foundations skills:

- **Unit 7:** requirements, examples, decision tables, algorithm boundaries, and explanations of practical cost;
- **Unit 8:** traceback reading, small reproductions, one-hypothesis-at-a-time debugging, and exception contracts;
- **Unit 9:** `pathlib`, encodings, JSON, context managers, and external-data validation;
- **Unit 10:** `src` layouts, packages, editable installation, virtual environments, and `pyproject.toml`;
- **Unit 11:** explicit collaborators, object responsibilities, and replacing a dependency through composition; and
- **Unit 12:** thin `main()` adapters, streams, exit status, configuration, logging, and one real subprocess check.

You do not need to memorize those units before continuing. Revisit a linked lesson when a boundary feels unfamiliar. Unit 14 will later combine tests with typing, formatting, linting, pre-commit, and broader quality gates; this unit stays focused on evidence about runtime behavior.

Prepare a project where pytest can discover the truth

The main workspace should be a local folder rather than only a notebook. Create this shape:

```text
meteor-watch-lab/
├── pyproject.toml
├── src/
│   └── meteor_watch/
│       ├── __init__.py
│       └── risk.py
└── tests/
    └── test_risk.py
```

Use the environment workflow from Unit 10. In an existing checkout of this course, the development dependencies already provide pytest and pytest-cov. For an independent practice project, install the testing tools into that project's virtual environment:

```bash
python -m pip install -e .
python -m pip install pytest pytest-cov hypothesis
python -m pytest --version
```

`python -m pytest` makes the interpreter choice explicit. If `pytest` and `python -m pytest` report different installations, stop and inspect the active environment before interpreting any failure.

Create a tiny first check in `tests/test_environment.py`:

```python
import meteor_watch


def test_package_imports_from_the_practice_environment():
    assert meteor_watch.__name__ == "meteor_watch"
```

Then run it from the directory containing `pyproject.toml`:

```bash
python -m pytest -q tests/test_environment.py
```

Expected evidence is one collected and passing test. A missing import is not a reason to edit the assertion; it is evidence about installation or layout.

Use notebooks for direct experiments

Colab and JupyterLab are still useful for calling pure functions, constructing case tables, and studying assertion behavior. Real discovery, command selection, installed imports, coverage, and child-process checks are clearest in a project terminal. The lessons label which boundary each exercise needs. :::

Keep a testing evidence log

For each lab, record five short fields:

Field	Example
Contract	`70` belongs to the high-risk interval.

Field	Example
Prediction	The focused test will fail with `medium` versus `high`.
Observation	Call phase failed at `assert risk_level(70) == "high"`.
One change	Change the second comparison from `<= 70` to `< 70`.
Clean result	Focused node and complete suite both pass.

This log prevents a common testing mistake: changing several inputs, assertions, and production lines until green appears without knowing which cause mattered. A green run whose history you cannot explain is weaker evidence than a deliberate red/green pair.

Plan the active work

Budget approximately **30–42 hours** for setup, typing, prediction, command runs, quizzes, repair clinics, final labs, and the challenge. A practical pace is:

1. spend one session on behavior-focused assertions;
2. give discovery and failure reading a full session in a clean local project;
3. separate parametrization and fixtures across different days;
4. practise dependency control after restarting the interpreter so old patches or mock state cannot mislead you;
5. reserve one session for test boundaries and coverage;
6. install Hypothesis and complete generated-case work in a fresh environment; and
7. attempt the challenge later without copying Meteor Watch tests.

Suggested stopping points appear at the final lab of each lesson. You are ready to continue when you can explain the evidence you kept, not merely when a green line appears.

Begin with the behavior worth preserving

Continue to Write Tests That Protect Behavior. Keep the wind-risk contract nearby and ask what each test can honestly claim. :::

Write Tests That Protect Behavior

Online lesson and runnable notebook

Turn public behavior into independent pytest assertions, expose false confidence and brittle checks, and build a focused protection map for a weather-risk rule.

- **Level:** Python Foundations
- **Estimated time:** 3–4 hours
- **You will learn:** Translate a behavioral contract into independent assertions, recognize tests that pass without protecting the right behavior, and keep checks stable across safe refactoring.
- **Practice in:** A local pytest project, with optional notebook experiments for direct function calls

Meteor Watch receives a wind speed and returns one of three labels. The first version appears reasonable:

```python
def risk_level(wind_kph):
    if wind_kph < 40:
        return "low"
    if wind_kph < 70:
        return "medium"
    return "high"
```

Running the function with `25` displays `"low"`. That is an example, but it is not yet a useful protection net. Which threshold should be included in medium? What should happen for a negative speed? Would changing `70` to `71` be caught? A test becomes meaningful when its expected result comes from an explicit contract and the failure would identify a behavior worth preserving.

As you work, answer these questions with code and failure evidence:

- What exactly does one passing test support, and what remains unknown?
- How can Arrange–Act–Assert make the behavior visible without adding ceremony?
- Which assertions observe public results rather than implementation trivia?
- How do false positives and false negatives create different kinds of risk?
- Which ordinary, boundary, and invalid cases protect the wind rule?

1. Start with a contract, not the current implementation

Use this contract throughout the lesson:

Input	Required behavior
a real number below `0`	raise `ValueError` with `wind must be non-negative`
`0 <= wind < 40`	return `"low"`
`40 <= wind < 70`	return `"medium"`
`wind >= 70`	return `"high"`

The table supplies expected values independently of the function body. A test for `70` should expect `"high"` because the published interval begins at 70, not because the current code uses a particular comparison.

A minimal pytest test is an ordinary function whose name begins with `test_`:

```python
from meteor_watch.risk import risk_level


def test_seventy_kph_begins_high_risk():
    observed = risk_level(70)
    assert observed == "high"
```

The lines have three jobs:

1. **Arrange:** choose `70` because the contract makes it a boundary.
2. **Act:** call `risk_level(70)` and retain the observed result.
3. **Assert:** compare the observation with the contract's `"high"`.

The test name records the scenario and promised result. If it fails in a list of hundreds, `test_seventy_kph_begins_high_risk` is more useful than `test_risk_2`.

Arrange–Act–Assert describes a reading pattern. It does not require three comments in every five-line test. Blank lines or well-chosen names can make the stages clear:

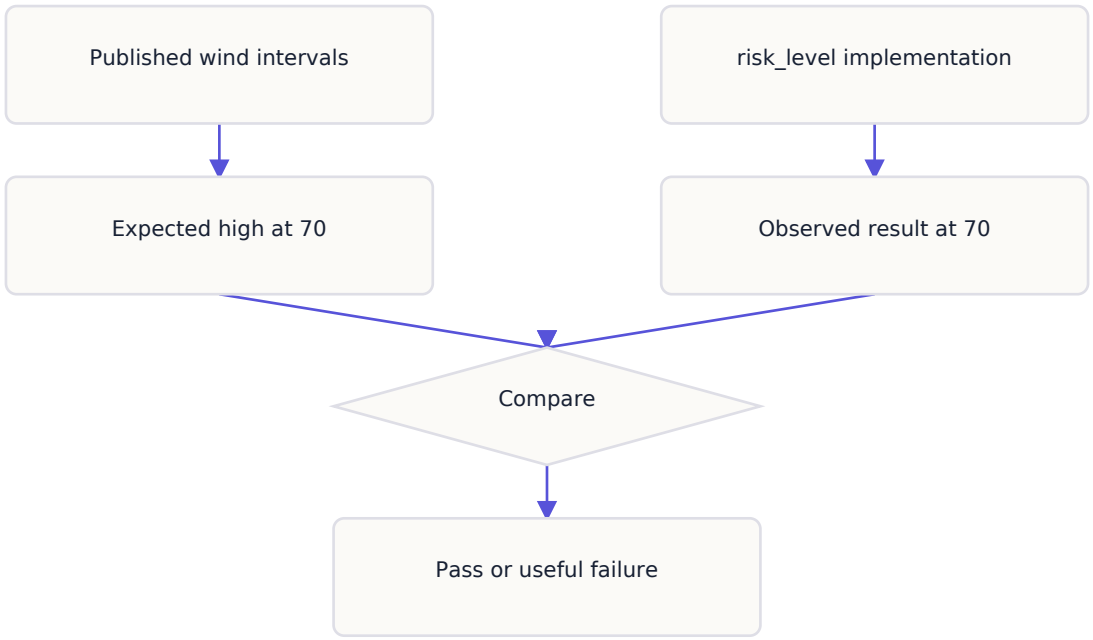
```python
def test_calm_wind_is_low_risk():
    observed = risk_level(12)

    assert observed == "low"
```

Predict a deliberate boundary defect

Change the production comparison to `if wind_kph <= 70:`. Predict the value for `70`, then run only the boundary test. It should fail with `"medium"` versus `"high"`. Restore `< 70` and run again. Keep both outputs: the red run shows that the test can notice the defect; the green run shows that this selected case now matches the contract.

The contract supplies the expected result. Copying a decision from the implementation back into the test would make both sides share the same mistake.



2. State one observable promise per test

A focused test may contain several assertions when they describe one result. For example, a parsed observation promises a station, wind value, and note as one returned record:

PYTHON
```
def parse_observation(line):
    station, wind_text, note = line.split("|", maxsplit=2)
    return {
        "station": station.strip(),
        "wind_kph": float(wind_text),
        "note": note.strip(),
    }


def test_parse_observation_returns_the_three_public_fields():
    observed = parse_observation(" ridge-7 | 42.5 | crosswind ")

    assert observed["station"] == "ridge-7"
    assert observed["wind_kph"] == 42.5
    assert observed["note"] == "crosswind"
```

All three assertions explain one public record. By contrast, placing parsing, risk classification, file writing, and notification in one test gives a failure too many possible causes. Test names and boundaries should make the broken promise easy to locate.

Compare direct and vague assertions

This assertion technically passes for any valid risk label:

PYTHON
```
def test_risk_returns_text():
    assert isinstance(risk_level(70), str)
```

It does not protect the important result. An implementation that always returns `"low"` still passes. The direct equality assertion is stronger:

```python
def test_seventy_kph_begins_high_risk():
    assert risk_level(70) == "high"
```

Prefer the clearest assertion that expresses the contract. Pytest can explain plain `assert` expressions, so a custom `if` followed by `raise AssertionError` usually hides useful details.

Check exceptions as behavior

Invalid input is not protected by a test that merely calls the function and lets any exception end the test. State the promised type and stable part of the message:

```python
import pytest


def test_negative_wind_is_rejected():
    with pytest.raises(ValueError, match="wind must be non-negative"):
        risk_level(-0.1)
```

`pytest.raises` fails if no exception occurs or if the type differs. The `match` expression checks user-relevant context without coupling to an entire traceback. Do not catch `Exception` in the production function merely to make this test pass; the contract names `ValueError` because the value is outside the accepted domain.

Compare measurements approximately

Binary floating-point operations can produce a nearby value instead of an exact decimal. If Meteor Watch converts 10 metres per second to 36 kilometres per hour, the contract is numeric closeness rather than a formatted string:

```python
import pytest


def metres_per_second_to_kph(speed):
    return speed * 3.6


def test_speed_conversion_is_numerically_close():
    assert metres_per_second_to_kph(10 / 3) == pytest.approx(12.0)
```

Use exact equality when exactness is the behavior: category labels, integer counts, dictionary keys, and normalized text should not be made fuzzy.

Checkpoint: assertions that describe behavior

Connect assertions to the wind contract

Choose the assertion that gives the clearest evidence.

1. The contract says high risk begins at 70 kph. Which assertion protects that boundary most directly?

- **A.** assert risk_level(70) == "high"

- **B.** assert isinstance(risk_level(70), str)
- **C.** assert risk_level(70)
- **D.** assert risk_level(70) in {"low", "medium", "high"}

2. Which test checks both the promised exception type and its useful context for wind -1?

- **A.** Call risk_level(-1) without an assertion
- **B.** Use pytest.raises(ValueError, match="non-negative")
- **C.** Use pytest.raises(Exception) without a match
- **D.** Assert that risk_level(-1) is None

3. Which result is the best fit for pytest.approx?

- **A.** The exact label "medium"
- **B.** The integer station count 4
- **C.** A converted measurement expected to be near 12.0
- **D.** The dictionary key "wind_kph"

Write your choices before checking the answer key at the end of this book.

3. Separate examples, checks, and test suites

An interactive example answers, “What happened in this run?”

```
PYTHON
print(risk_level(25))
```

An assertion inside a notebook stops when a claim is false:

```
PYTHON
assert risk_level(25) == "low"
```

A pytest test adds discovery, a stable name, independent execution, rich failure reporting, selection, and inclusion in a repeatable suite:

```
PYTHON
def test_twenty_five_kph_is_low_risk():
    assert risk_level(25) == "low"
```

All three can help, but they provide different evidence. A copied screenshot of one green run does not guarantee that another machine can discover the same test. A suite becomes dependable when the project declares its dependencies, tests run from a clean state, and each test controls the resources it needs.

4. Recognize false confidence

A **false positive** in everyday testing language is a test that passes while the behavior you intended to protect is wrong. Consider four examples.

A test with no assertion

```python
def test_risk_example_runs():
    risk_level(70)
```

This detects an unexpected exception for `70`, but it cannot distinguish `"high"` from `"medium"`. That may be a valid smoke check only if “does not raise” is the written promise. Its current name does not say so.

An assertion on the arranged value

```python
def test_high_wind():
    wind = 70
    observed = risk_level(wind)

    assert wind == 70
```

The test never checks `observed`. It proves that the test assigned `70` to `wind`, not that the system classified it.

Expected logic copied from production

```python
def test_risk_matches_expected_formula():
    wind = 70
    if wind < 40:
        expected = "low"
    elif wind < 70:
        expected = "medium"
    else:
        expected = "high"

    assert risk_level(wind) == expected
```

This case currently works, but duplicating every branch makes the test capable of sharing the same typo as the implementation. Prefer expected values from a small contract table. A separate simpler model can be a valid oracle later, but it must actually be independent and easier to trust.

An assertion too broad to notice the defect

```python
def test_risk_is_a_known_label():
    assert risk_level(70) in {"low", "medium", "high"}
```

This protects a useful invariant—no unknown label—but not the boundary. Keep it only if that invariant has value alongside exact examples, not instead of them.

Make the test red before trusting a new check

Temporarily introduce the defect the test is meant to catch, or write the test before the missing behavior. If the test stays green, inspect whether it calls the right code and asserts the right observation. Restore production code and rerun the complete suite afterward. :::

5. Avoid tests that reject safe refactoring

A **false negative** is a test that fails even though the public behavior is still acceptable. These tests slow improvement because ordinary refactoring looks like a defect.

Suppose the first implementation has a private helper:

```python
def _upper_boundary_for(label):
    boundaries = {"low": 40, "medium": 70}
    return boundaries[label]


def risk_level(wind_kph):
    if wind_kph < _upper_boundary_for("low"):
        return "low"
    if wind_kph < _upper_boundary_for("medium"):
        return "medium"
    return "high"
```

This test couples to that private shape:

```python
def test_medium_private_boundary():
    assert _upper_boundary_for("medium") == 70
```

Refactoring to a tuple of ranges removes the helper and breaks the test even if every promised result remains correct. The public-boundary test survives:

```python
def test_seventy_kph_begins_high_risk():
    assert risk_level(70) == "high"
```

Private helpers can be tested directly when they contain complicated, independently meaningful behavior. The warning is not “never test a helper.” It is “do not make incidental structure part of the contract by accident.”

The same caution applies to call counts. If the contract says “send one public alert,” an exactly-once interaction may matter. If the function is free to cache, batch, or reorganize an internal helper, an exact call count is trivia.

Checkpoint: identify misleading green and red results

Find false confidence and brittle checks

Judge what each test actually observes.

1. A test calls observed = risk_level(70) but asserts only wind == 70. What is the main defect in the test?

- **A.** It uses too few local variables
- **B.** It should compare floating point approximately
- **C.** It must call a private helper
- **D.** It never asserts the system’s observed result

2. Why can a test of _upper_boundary_for become brittle?

- **A.** A safe refactor may remove the private helper while preserving risk_level behavior
- **B.** Private functions can never return values
- **C.** Pytest refuses to import names beginning with an underscore
- **D.** Boundary values cannot be asserted

3. What does assert risk_level(70) in {"low", "medium", "high"} establish?

- **A.** That 70 is high risk
- **B.** Only that the result is one of the known labels
- **C.** That every numeric input is classified correctly
- **D.** That no exception can occur for any value

Write your choices before checking the answer key at the end of this book.

6. Choose examples by input class and boundary

You cannot test every real number. Instead, partition the contract into groups that should behave alike:

Class	Representative values	Why they matter
invalid negative	`-0.1`	closest invalid value below zero
low	`0`, `12`, `39.9`	lower boundary, ordinary case, upper edge
medium	`40`, `55`, `69.9`	lower boundary, ordinary case, upper edge
high	`70`, `120`	lower boundary and ordinary high value

Start with one ordinary case in each valid class, then test both sides of every boundary. If input arrives as text, that parser adds separate empty, malformed, and non-finite cases. Do not silently mix those parser requirements into the numeric classifier’s tests.

A compact suite can still be explicit:

PYTHON
```
import pytest

from meteor_watch.risk import risk_level


def test_zero_kph_is_low_risk():
    assert risk_level(0) == "low"


def test_just_below_forty_is_low_risk():
    assert risk_level(39.9) == "low"


def test_forty_begins_medium_risk():
    assert risk_level(40) == "medium"


def test_just_below_seventy_is_medium_risk():
```

PYTHON (CONTINUED)

```
    assert risk_level(69.9) == "medium"


def test_seventy_begins_high_risk():
    assert risk_level(70) == "high"


def test_negative_wind_is_rejected():
    with pytest.raises(ValueError, match="wind must be non-negative"):
        risk_level(-0.1)
```

Lesson 3 will turn related cases into a parametrized table. First learn to justify each row. Parametrization makes a design concise; it does not decide which cases belong in the design.

7. Keep tests independent and deterministic

Each test should be able to run alone and in any order. This pair is not independent:

PYTHON

```
observed_labels = []


def test_record_low_label():
    observed_labels.append(risk_level(10))
    assert observed_labels == ["low"]


def test_record_high_label():
    observed_labels.append(risk_level(90))
    assert observed_labels == ["low", "high"]
```

The second test passes only if the first test has already changed global state. Run it alone and it fails. Reverse the order and both may fail. Use fresh local state when shared history is not the behavior:

PYTHON

```
def test_records_low_label():
    observed_labels = []
    observed_labels.append(risk_level(10))
    assert observed_labels == ["low"]


def test_records_high_label():
    observed_labels = []
    observed_labels.append(risk_level(90))
    assert observed_labels == ["high"]
```

Current time, unseeded randomness, current directory, environment variables, network services, and leftover files create similar hidden dependencies. Later lessons provide fixtures and seams for them. For now, ask of every test: “What must already be true outside this function?” If the answer is not visible in the arrangement, isolation is at risk.

8. Explain the limit of every green result

After the six wind tests pass, accurate statements include:

- the selected boundary and ordinary values matched the stated intervals;
- the selected negative value raised the stated exception in this environment;

- the test suite did not observe shared state or real external services; and
- the deliberate `<= 70` defect was noticed by the 70-kph test.

Inaccurate statements include:

- `risk_level` has no bugs;
- every possible float behaves correctly;
- parsing, report writing, the CLI, and installation work;
- the suite will never be flaky; or
- high coverage has proven the algorithm.

Precise limits do not make tests weak. They make evidence honest and show what to investigate next.

Checkpoint: select a protection map

Choose cases and state honest conclusions

Use the wind intervals and test-isolation rules.

1. Which three values best reveal both sides of the 40-kph boundary?

- **A.** 10, 20, 30
- **B.** 40, 50, 60
- **C.** 39.9, 40, 40.1
- **D.** 70, 80, 90

2. A test passes only after another test appends to a module-level list. What is the best diagnosis?

- **A.** The assertion needs pytest.approx
- **B.** The function name is too descriptive
- **C.** The list must be made private
- **D.** The tests share order-dependent mutable state

3. Six selected wind examples pass. Which conclusion is justified?

- **A.** Those selected inputs produced the contract’s expected results in this run
- **B.** Every possible Python number is correct
- **C.** The CLI and package installation are proven
- **D.** No future refactor can introduce a defect

Write your choices before checking the answer key at the end of this book.

9. Build a protection map for Meteor Watch

Create `src/meteor_watch/risk.py` with this deliberately incomplete function:

```python
def risk_level(wind_kph):
    """Return low, medium, or high for a non-negative wind speed."""
```

PYTHON (CONTINUED)

```
    if wind_kph < 40:
        return "low"
    if wind_kph < 70:
        return "medium"
    return "high"
```

Create `tests/test_risk.py`. Your finished lab should:

1. write one ordinary case for each valid interval;
2. protect `0`, `40`, and `70` plus the values just below the latter two boundaries;
3. require `ValueError("wind must be non-negative")` for `-0.1`;
4. temporarily introduce one comparison defect and capture the focused red report;
5. repair the implementation rather than changing a correct expected value;
6. run the focused node and then the complete file from a clean process; and
7. write two sentences: what the suite supports and one risk it does not cover.

The negative rule requires this production guard:

PYTHON

```
def risk_level(wind_kph):
    """Return low, medium, or high for a non-negative wind speed."""
    if wind_kph < 0:
        raise ValueError("wind must be non-negative")
    if wind_kph < 40:
        return "low"
    if wind_kph < 70:
        return "medium"
    return "high"
```

Run:

BASH

```
python -m pytest -q tests/test_risk.py
```

Your observable result should name every case and finish green. Do not add parser, file, or CLI checks; they are different contracts.

Hint 1: turn the interval table into names

Start with `test_forty_begins_medium_risk` and `test_just_below_forty_is_low_risk`. Repeat the pattern around 70. The name should tell you which expected value belongs in the assertion.

Hint 2: prove the negative path fails for the right reason

Use `with pytest.raises(ValueError, match="non-negative"):` around one call. Run that node before adding the production guard; it should fail because no exception was raised.

Hint 3: find accidental coupling

Search the tests for globals, current time, random values, files, or one test calling another. This lab needs none of those dependencies; every case can call the public function with one explicit number.

Show one complete protection map

```python
import pytest

from meteor_watch.risk import risk_level


def test_zero_kph_is_low_risk():
    assert risk_level(0) == "low"


def test_ordinary_low_wind_is_low_risk():
    assert risk_level(18) == "low"


def test_just_below_forty_is_low_risk():
    assert risk_level(39.9) == "low"


def test_forty_begins_medium_risk():
    assert risk_level(40) == "medium"


def test_ordinary_medium_wind_is_medium_risk():
    assert risk_level(55) == "medium"


def test_just_below_seventy_is_medium_risk():
    assert risk_level(69.9) == "medium"


def test_seventy_begins_high_risk():
    assert risk_level(70) == "high"


def test_ordinary_high_wind_is_high_risk():
    assert risk_level(110) == "high"


def test_negative_wind_is_rejected():
    with pytest.raises(ValueError, match="wind must be non-negative"):
        risk_level(-0.1)
```

One honest conclusion is: “These selected examples protect both sides of the documented numeric thresholds and one negative value. They do not test parsing, non-finite numbers, files, notifications, or the process boundary.”

Key points

- Derive expected values from a behavioral contract, not from the current implementation.
- A focused test arranges visible state, performs one meaningful action, and asserts an observable promise.
- A green test supports only the case and environment it actually exercised.
- Broad, vacuous, wrong-value, and copied-logic assertions can pass without protecting the intended behavior.
- Tests of incidental helpers or call sequences can reject safe refactoring.
- Choose ordinary, boundary, invalid, and regression examples deliberately.
- Run a new test red for the expected reason before trusting its green result.
- Keep tests independent, deterministic, and able to run alone.

Continue learning

- pytest documentation: get started with plain assertions
- pytest documentation: assertion introspection, exceptions, and approximate comparisons
- Python documentation: floating-point limitations
- Next: Run Pytest and Read the Failure

Run Pytest and Read the Failure

Online lesson and runnable notebook

Build a discoverable src-layout pytest project, select precise nodes, distinguish collection and execution phases, and turn assertion reports into one-cause-at-a-time repairs.

- **Level:** Python Foundations
- **Estimated time:** 4–5 hours
- **You will learn:** Make pytest discover the intended project, select one test precisely, classify where a run failed, and use rich assertion evidence to repair the earliest useful cause.
- **Practice in:** A local `src` project and terminal; a notebook can reproduce direct assertions but not the complete discovery and process workflow

A good assertion is useful only if the test runner can find it, import the intended package, create its dependencies, and report the observation clearly. This lesson turns the small Meteor Watch checks into a real project and treats pytest output as structured debugging evidence.

Keep these questions visible:

- How does pytest turn directories, files, classes, functions, and parameter cases into collected node IDs?
- Which commands select exactly the evidence you need without hiding context?
- Did the run fail during collection, setup, the test call, or teardown?
- What do the traceback, assertion explanation, captured output, and summary each contribute?
- How can you repair one failure without allowing later noise to lead you away from the earliest cause?

1. Give production code and tests different homes

Create this project:

```text
meteor-watch-lab/
├── pyproject.toml
├── src/
│   └── meteor_watch/
│       ├── __init__.py
│       └── risk.py
└── tests/
    ├── test_import.py
    └── test_risk.py
```

The `src` layout makes an important mistake visible: importing code merely because the current directory happens to contain a package. Install the project into its virtual environment and let tests import it by its public package name.

Use a minimal `pyproject.toml`:

```toml
[build-system]
requires = ["setuptools>=68"]
build-backend = "setuptools.build_meta"

[project]
name = "meteor-watch"
version = "0.1.0"
requires-python = ">=3.10"

[tool.pytest.ini_options]
testpaths = ["tests"]
addopts = "-ra"
```

`testpaths` tells pytest where ordinary suite discovery begins. `-ra` adds a short summary for outcomes such as skips and expected failures. Configuration should make the ordinary command predictable; it should not hide warnings or force a maze of plugins.

Place the public function in `src/meteor_watch/risk.py`:

```python
def risk_level(wind_kph):
    """Return low, medium, or high for a non-negative wind speed."""
    if wind_kph < 0:
        raise ValueError("wind must be non-negative")
    if wind_kph < 40:
        return "low"
    if wind_kph < 70:
        return "medium"
    return "high"
```

Re-export it in `src/meteor_watch/__init__.py`:

```python
from .risk import risk_level

__all__ = ["risk_level"]
```

Add `tests/test_import.py`:

```python
import meteor_watch


def test_installed_package_exposes_risk_level():
    assert callable(meteor_watch.risk_level)
```

Create and activate an environment using the Unit 10 instructions for your platform, then run:

```bash
python -m pip install -e .
python -m pytest -q
```

The editable install means changes under `src/` are visible without rebuilding a wheel. It does not mean “import any file near the terminal.” Confirm the interpreter and imported location when results are surprising:

```bash
python -c "import sys, meteor_watch; print(sys.executable); print(meteor_watch.__file__)"
```

Do not repair an import error by editing a correct assertion

If collection cannot import `meteor_watch`, pytest has not called the test. Check the active interpreter, editable installation, package directory, and module spelling before changing expected values. :::

2. Follow discovery from a path to a node ID

With default conventions, pytest searches configured directories for files named `test_*.py` or `*_test.py`. It then collects test functions and methods whose names begin with `test`. A class can group tests when its name begins with `Test` and it does not define its own constructor.

This file yields three function nodes:

```python
from meteor_watch import risk_level


def test_calm_wind_is_low():
    assert risk_level(12) == "low"


def test_forty_begins_medium():
    assert risk_level(40) == "medium"


class TestHighRisk:
    def test_seventy_begins_high(self):
        assert risk_level(70) == "high"
```

Ask pytest to show the collection without running the bodies:

```bash
python -m pytest --collect-only -q
```

Representative output:

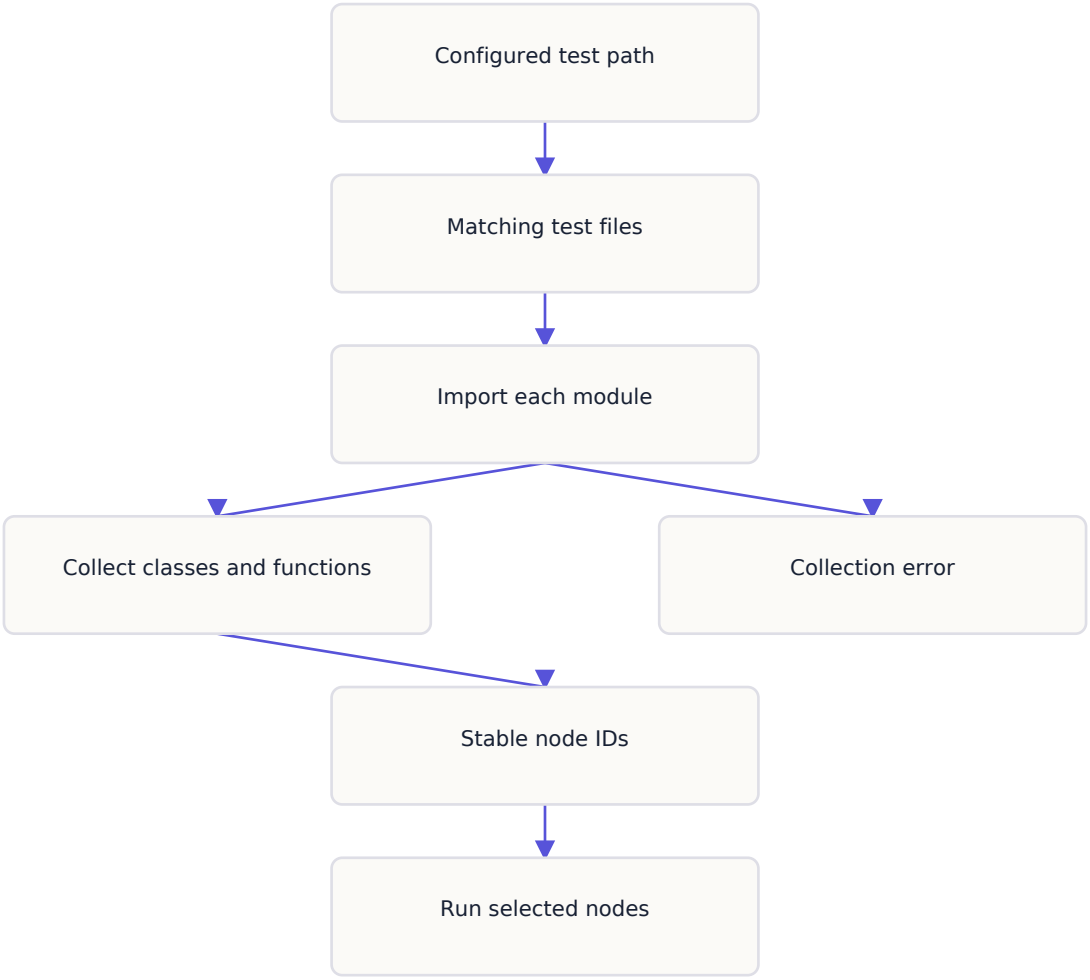
```text
tests/test_import.py::test_installed_package_exposes_risk_level
tests/test_risk.py::test_calm_wind_is_low
tests/test_risk.py::test_forty_begins_medium
tests/test_risk.py::TestHighRisk::test_seventy_begins_high

4 tests collected in 0.02s
```

Each `::` adds another collected level. The complete string is a **node ID**. It is both a location and a precise selector:

```bash
python -m pytest -q tests/test_risk.py::TestHighRisk::test_seventy_begins_high
```

Pytest first discovers candidates and imports modules. Only successfully collected nodes can enter setup, call, and teardown.



Import each module → Collect classes and functions: import succeeds

Import each module → Collection error: import fails

Diagnose a test that never appears

This function is valid Python but is not collected:

```python
def checks_medium_boundary():
    assert risk_level(40) == "medium"
```

Rename it `test_medium_boundary`. Do not add a manual call at the bottom of the file; that would run during import and blur collection with execution.

Likewise, `risk_checks.py` is not a conventional test filename. Discovery conventions make a suite predictable for maintainers and tools. A custom pattern is possible, but changing configuration to accommodate one accidental name is usually less clear than naming the file `test_risk.py`.

3. Select enough evidence, not more noise

Use the broadest command that still answers the current question:

Command	Useful question
`python -m pytest`	Does the complete configured suite pass?
`python -m pytest -q`	Can I see a compact complete result?
`python -m pytest tests/test_risk.py`	What happens in this file?
`python -m pytest tests/test_risk.py::test_forty_begins_medium`	What happens in this exact node?
`python -m pytest -k boundary`	Which collected nodes with matching names pass?
`python -m pytest -x`	What is the first failure in this run?
`python -m pytest --maxfail=2`	What are the first two failures before stopping?
`python -m pytest --lf`	Do nodes that failed in the previous run now pass?
`python -m pytest -vv`	Which full node IDs and parameter IDs are running?

`-k` matches collected names and parents; it is not a text search through source. Quote complex expressions in the shell, such as `-k "boundary and not negative"`.

`-x` reduces noise while debugging the earliest failure. It is not a final verification command. After the focused node turns green, run its file and then the complete suite. A local repair may expose a second failure or accidentally break another contract.

Checkpoint: discover and select the intended node

Follow pytest discovery and selection

Choose the command or diagnosis that matches the project evidence.

1. Which command runs only test_seventy_begins_high inside class TestHighRisk in tests/test_risk.py?

- **A.** python -m pytest -k tests/test_risk.py
- **B.** python -m pytest tests/test_risk.py::TestHighRisk::test_seventy_begins_high
- **C.** python tests/test_risk.py::TestHighRisk
- **D.** python -m pytest –collect-only TestHighRisk

2. A valid function named checks_medium_boundary does not appear under –collect-only. What is the most direct repair?

- **A.** Add assert True at module scope
- **B.** Call the function while the module imports
- **C.** Rename it so the function name begins with test_
- **D.** Change the production result

3. What is the best use of pytest -x during a repair?

- **A.** Prove that no later failures exist
- **B.** Skip every slow test permanently

- **C.** Rewrite all expected values automatically
- **D.** Stop after the first failure so its evidence can be investigated without later noise

Write your choices before checking the answer key at the end of this book.

4. Classify the phase before changing code

For a collected test, pytest can run three phases:

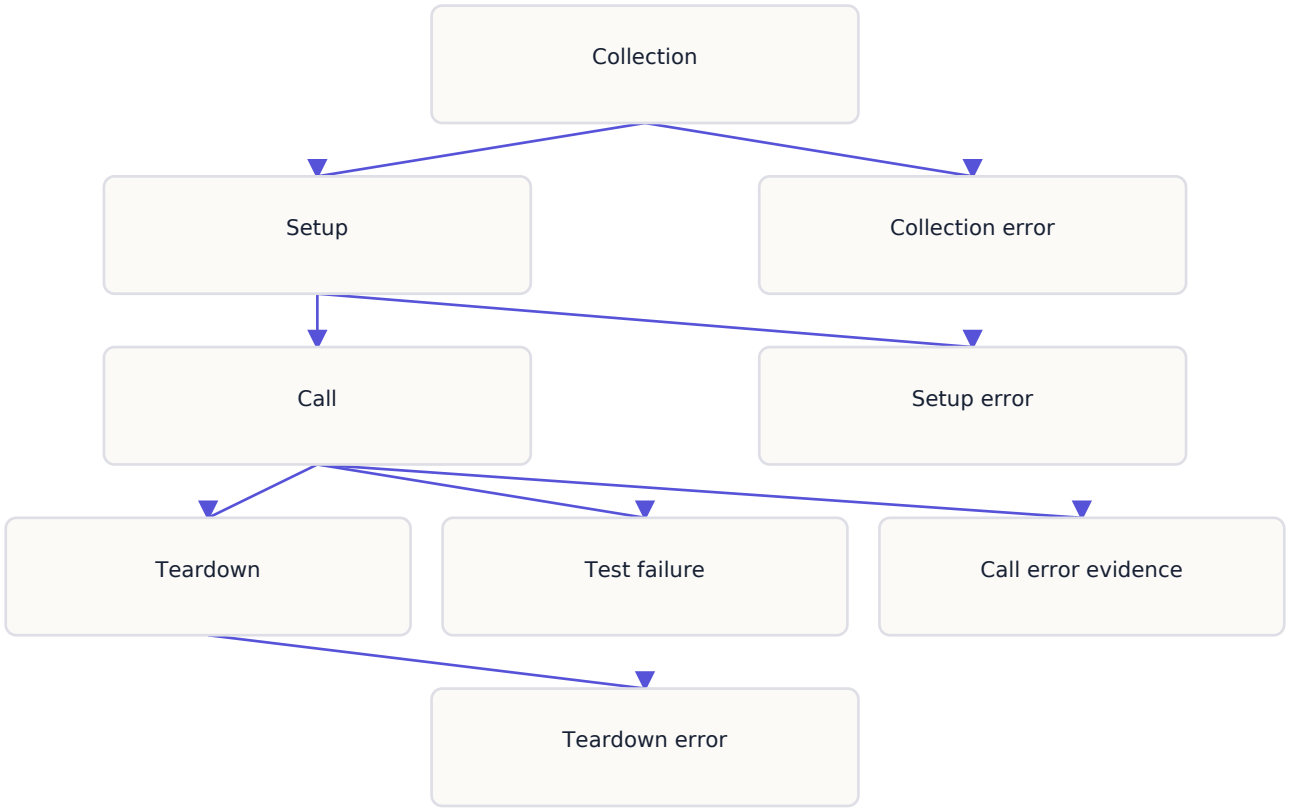
1 **setup** obtains fixtures and prepares the test world;

2 **call** executes the test function; and

3 **teardown** releases fixture-managed resources.

An assertion mismatch during the call phase is reported as a **failure**. An unexpected exception during fixture setup or teardown is usually reported as an **error**. Collection errors happen earlier, before those phases exist.

The report phase narrows the search. Editing the classifier cannot repair a misspelled fixture or a module that never imported.



Collection → Collection error: import or syntax problem

Setup → Setup error: fixture cannot prepare

Call → Test failure: assertion is false

Call → Call error evidence: unexpected exception

Teardown → Teardown error: cleanup fails

Consider a misspelled fixture request:

```python
def test_report_uses_station(station_reccord):
    assert station_reccord["station"] == "ridge-7"
```

Pytest can collect the function, but setup cannot find `station_reccord`. The report includes `fixture 'station_reccord' not found` and lists available fixtures. The production parser never ran. Rename the argument or define the intended fixture; do not change parsing code.

By contrast, this test reaches the call and observes a wrong result:

```python
def test_forty_begins_medium():
    observed = risk_level(40)
    assert observed == "medium"
```

If `observed` is `"low"`, inspect the 40-kph comparison or the contract. The phase and compared values point to a much smaller investigation.

5. Read every useful part of an assertion report

Introduce this defect temporarily:

```python
def risk_level(wind_kph):
    if wind_kph < 0:
        raise ValueError("wind must be non-negative")
    if wind_kph <= 40:
        return "low"
    if wind_kph < 70:
        return "medium"
    return "high"
```

Run the focused node without `-q`:

```bash
python -m pytest tests/test_risk.py::test_forty_begins_medium
```

A report resembles:

```text
============================= test session starts ==============================
platform linux -- Python 3.12.4, pytest-9.1.1
rootdir: /work/meteor-watch-lab
configfile: pyproject.toml
collected 1 item

tests/test_risk.py F                                                     [100%]

=================================== FAILURES ===================================
_____________________________ test_forty_begins_medium _____________________________

    def test_forty_begins_medium():
        observed = risk_level(40)
>       assert observed == "medium"
E       AssertionError: assert 'low' == 'medium'
E
E         - medium
E         + low
```

TEXT (CONTINUED)

```
tests/test_risk.py:11: AssertionError
========================== short test summary info ==========================
FAILED tests/test_risk.py::test_forty_begins_medium - AssertionError: ...
============================== 1 failed in 0.06s ===============================
```

Annotate it:

- **session header:** interpreter, pytest version, root directory, and selected configuration identify the environment;
- **collection:** one item confirms the intended node existed;
- **failure heading and node:** identify the scenario;
- **source excerpt and `>`:** show the exact expression that was false;
- `E` **lines:** show the compared values and pytest's explanation;
- **location:** points to the test assertion, from which the call can be traced;
- **short summary:** collects every non-passing node for navigation; and
- **duration/result:** confirms the run ended with one failure rather than a collection interruption.

The report does not order you to change `low` to `medium` in production. It states the observation. The interval contract decides the correct repair: change `<= 40` to `< 40`.

6. Let plain assertions produce useful diffs

Pytest rewrites assertions as test modules are imported, retaining information about subexpressions. Use direct comparisons so that information stays visible.

Strings reveal the changed region

PYTHON

```
def format_alert(station, level):
    return f"{station}: {level.upper()} risk"


def test_alert_format_names_station_and_level():
    assert format_alert("ridge-7", "high") == "ridge-7: HIGH risk"
```

A misspelled suffix produces a character-level diff. Avoid replacing this with `assert "HIGH" in observed` unless the rest of the format truly does not matter.

Lists reveal order and missing elements

```python
def risky_stations(observations):
    return [
        item["station"]
        for item in observations
        if item["risk"] == "high"
    ]


def test_risky_stations_preserve_observation_order():
    observations = [
        {"station": "ridge-7", "risk": "high"},
        {"station": "lake-2", "risk": "low"},
        {"station": "tower-1", "risk": "high"},
    ]

    assert risky_stations(observations) == ["ridge-7", "tower-1"]
```

If order is not a promise, compare a set. If order is a promise, converting both sides to sets creates a false positive.

Dictionaries reveal differing fields

```python
def test_parsed_observation_has_expected_fields():
    observed = parse_observation("ridge-7|42.5|crosswind")

    assert observed == {
        "station": "ridge-7",
        "wind_kph": 42.5,
        "note": "crosswind",
    }
```

One dictionary equality gives pytest enough information to show missing, extra, or differing entries while preserving the whole record contract.

Checkpoint: classify and read reports

Use the failure phase and assertion evidence

Choose the diagnosis supported by the report.

1. Pytest reports ModuleNotFoundError while importing tests/test_risk.py and collects zero items. What should be investigated first?

- **A.** The active environment, editable install, and import spelling
- **B.** The expected label in the boundary assertion
- **C.** The fixture teardown order
- **D.** The branch coverage percentage

2. The report says fixture "station_reccord" not found during setup. Which change best follows the evidence?

- **A.** Return a different risk label
- **B.** Correct the requested fixture name or define the fixture
- **C.** Use pytest.approx on the station name

- **D.** Ignore the error and run coverage

3. Why is direct dictionary equality useful for a record contract?

- **A.** It prevents pytest from showing values
- **B.** It ignores all extra keys
- **C.** Pytest can explain missing, extra, and differing fields
- **D.** It automatically changes production data

Write your choices before checking the answer key at the end of this book.

7. Assert anticipated exceptions precisely

Use `pytest.raises` as a context around only the action expected to fail:

```python
import pytest


def parse_wind(text):
    try:
        wind = float(text)
    except ValueError as error:
        raise ValueError(f"invalid wind: {text!r}") from error
    if wind < 0:
        raise ValueError("wind must be non-negative")
    return wind


def test_parse_wind_rejects_non_numeric_text():
    with pytest.raises(ValueError, match="invalid wind") as error_info:
        parse_wind("east")

    assert "'east'" in str(error_info.value)
```

Code after the call belongs outside the `with` block. If you place another operation inside, an exception from that operation can satisfy a broad context and create false confidence.

Use a narrow message regular expression. User-specific values can be checked through `error_info.value`. Do not assert the entire traceback or platform path. Those are report details rather than the exception contract.

8. Compare numeric results with an explicit tolerance

`pytest.approx` is asymmetric in how it treats expected and observed values, but ordinary use reads naturally:

```python
import pytest


def average_wind(values):
    return sum(values) / len(values)


def test_average_wind_is_close_to_expected_measurement():
    observed = average_wind([10.0, 10.1, 9.9])
    assert observed == pytest.approx(10.0, abs=0.001)
```

Choose a tolerance from the domain or numeric operation, not merely a large number that makes red disappear. If the requirement is “display one decimal place,” test the formatted string separately from the numeric calculation.

9. Use skip and expected failure deliberately

Sometimes a test cannot run in a known environment, or a documented defect is temporarily accepted. Pytest can represent those states:

```python
import sys

import pytest


@pytest.mark.skipif(sys.version_info < (3, 11), reason="requires tomllib")
def test_toml_configuration_example():
    import tomllib

    assert tomllib.loads("limit = 70")["limit"] == 70
```

An expected failure can use `@pytest.mark.xfail(reason="issue 42")`. Add `strict=True` when an unexpected pass should force review rather than silently remaining XPASS. These marks communicate a specific condition; they are not a bin for inconvenient failures. A failing boundary test whose contract is current should remain red until repaired.

10. Repair the earliest useful failure

Use this repeatable sequence:

1. run `--collect-only` if discovery or import is uncertain;
2. run with `-x` to stop at the earliest selected failure;
3. identify collection, setup, call, or teardown;
4. state one hypothesis that explains the exact report;
5. run the smallest node or collection command capable of testing it;
6. change one cause;
7. rerun the focused evidence;
8. rerun the file; and
9. rerun the complete suite from a clean process.

Changing production and expected values together destroys the red/green evidence. Likewise, deleting a test because it fails is not a repair unless the contract has intentionally changed and that decision is recorded.

Checkpoint: choose the next command

Choose the smallest useful rerun

Match the command to the uncertainty you need to resolve.

1. You renamed a test file and are unsure whether its functions are discovered. What should you run first?

- **A.** pytest –cov
- **B.** pytest –lf
- **C.** pytest -x after changing production
- **D.** python -m pytest –collect-only -q

2. A single boundary node failed and you changed one comparison. What is the best immediate rerun?

- **A.** That exact node ID
- **B.** Only the package import command
- **C.** Delete pytest's cache
- **D.** An unrelated fixture test

3. Why run the complete suite after the focused node turns green?

- **A.** To make pytest rewrite production code
- **B.** To detect interactions or regressions outside the repaired node
- **C.** To guarantee the program has no bugs
- **D.** To avoid installing the package

Write your choices before checking the answer key at the end of this book.

11. Run the mixed-failure weather clinic

Create a disposable copy of the Meteor Watch project. Add four faults, one in each category:

1. in `tests/test_import.py`, import `meteor_wotch` to create a collection error;
2. in one collected test, request a nonexistent `weather_record` fixture;
3. change the 40-kph comparison in production from `< 40` to `<= 40`;
4. write one expected dictionary with the wrong note.

Your task is not to repair all four immediately. Produce an evidence record:

Run	Phase	Earliest useful line	Hypothesis	One change	Result
1	collection	…	…	…	…
2	setup	…	…	…	…
3	call	…	…	…	…
4	call	…	…	…	…

Use `--collect-only`, `-x`, an exact node ID, and finally the complete suite. Annotate at least one report with the environment, node, source line, compared values, and summary.

Hint 1: zero collected items changes the first question

If module import fails, no fixture or assertion in that file can run. Correct the import spelling and rerun collection before reasoning about wind values.

Hint 2: distinguish fixture lookup from the test call

A missing fixture is setup evidence. Pytest lists available fixtures. Make the argument name match an existing fixture or add the small fixture the test actually needs.

Hint 3: do not change a contract-derived expected value

For the 40-kph failure, compare the production condition with `0 <= wind < 40`. For the dictionary failure, compare the expected note with the supplied input. Only one side should contradict the contract in each case.

Show a completed diagnosis sequence

One valid order is:

1. `--collect-only -q` exposes `ModuleNotFoundError: meteor_wotch`; correct the import and confirm nodes collect.
2. `-x` exposes `fixture 'weather_record' not found`; correct the requested fixture and rerun that node.
3. The next `-x` run reaches `test_forty_begins_medium` and reports `low` versus `medium`; restore `< 40` and rerun the exact node.
4. The dictionary test shows only the note differs; correct whichever side contradicts its written input/contract.
5. Run `python -m pytest` from a fresh process and retain the complete green summary.

The important artifact is the phase-based explanation, not merely four edited lines.

Key points

- A `src` layout plus editable installation makes import assumptions visible.
- Discovery creates stable node IDs from matching files, classes, functions, and later parameter cases.
- Select a suite, file, node, keyword, or previous failure based on the question you are answering.
- Collection, setup, call, and teardown evidence point to different causes.
- Plain assertions let pytest display useful scalar, text, sequence, and mapping differences.
- `pytest.raises` and `pytest.approx` express exception and numeric contracts precisely.
- Skip and xfail marks should state a real condition, never hide an unexplained failure.
- Repair one cause, rerun the focused evidence, then verify the complete suite.

Continue learning

- pytest documentation: discovery and the first test
- pytest documentation: test anatomy and execution phases
- pytest documentation: assertion explanations
- pytest documentation: good integration practices

- Next: Turn Examples into Clear Parametrized Cases

Turn Examples into Clear Parametrized Cases

Online lesson and runnable notebook

Partition a behavioral contract into representative and boundary cases, express them as readable pytest parameter nodes, and avoid hidden loops, copied oracles, mutation leaks, and combinatorial noise.

- **Level:** Python Foundations
- **Estimated time:** 3–4 hours
- **You will learn:** Derive a compact case table from input partitions, create useful parametrized node IDs, and keep every expected value and failure independently understandable.
- **Practice in:** A local pytest project for collection and failure IDs, with optional notebook work for designing case tables

The previous lesson produced several wind-risk tests with nearly identical shapes. Repetition is not automatically bad: every descriptive test made a boundary visible. Once the case design is sound, pytest parametrization can remove mechanical repetition while preserving a separate node and failure for each row.

This lesson begins before the decorator. You will decide which cases represent the contract, why each expected value is trustworthy, and how a failed case can identify itself without reading a list index.

Answer these questions as you build the suite:

- Which inputs should behave alike, and where does behavior change?
- Which just-below, exact, and just-above values expose a threshold?
- How does parametrization differ from a loop inside one test?
- What makes a parameter ID useful in a failure report?
- When should cases remain separate rather than join one complicated table?

1. Partition the input space before writing rows

Meteor Watch now derives an alert level from wind and visibility:

PYTHON

```
def alert_level(wind_kph, visibility_km):
    """Return green, amber, or red for valid weather measurements."""
    if wind_kph < 0:
        raise ValueError("wind must be non-negative")
    if visibility_km < 0:
        raise ValueError("visibility must be non-negative")
    if wind_kph >= 70 or visibility_km < 1:
        return "red"
    if wind_kph >= 40 or visibility_km < 5:
        return "amber"
    return "green"
```

The contract creates partitions:

- wind: invalid below 0, green contribution below 40, amber contribution from 40 through just below 70, and red contribution from 70 upward;
- visibility: invalid below 0, red contribution below 1, amber contribution from 1 through just below 5, and green contribution from 5 upward; and
- combination: the most severe triggered condition wins.

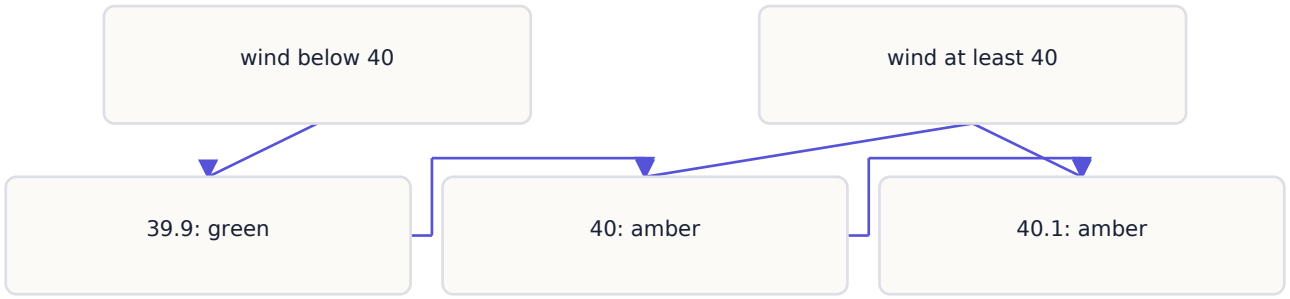
An **equivalence partition** is a group of inputs that should follow the same rule. Testing one representative can support that class, while boundaries need special attention because comparison operators decide which neighboring class owns the exact value.

Start with one variable at a time while keeping the other safely ordinary:

Case ID	Wind	Visibility	Expected	Reason
`calm-clear`	12	10	green	ordinary safe measurements
`wind-below-amber`	39.9	10	green	just below wind 40
`wind-at-amber`	40	10	amber	exact wind 40
`wind-below-red`	69.9	10	amber	just below wind 70
`wind-at-red`	70	10	red	exact wind 70
`visibility-below-green`	12	4.9	amber	just below visibility 5
`visibility-at-green`	12	5	green	exact visibility 5
`visibility-below-amber`	12	0.9	red	just below visibility 1
`visibility-at-amber`	12	1	amber	exact visibility 1

The IDs use the **resulting interval**, not an ambiguous phrase such as `visibility-boundary-2`. Read `visibility-at-green` as “at the lower boundary of the green visibility interval.” If that wording feels unclear to your team, prefer `visibility-five-is-green`. A useful ID needs no private legend.

Boundary values belong to one side of a comparison. The three observations around 40 reveal whether the implementation uses <, <=, or an incorrect threshold.



2. Turn the table into separate pytest nodes

`@pytest.mark.parametrize` supplies each row to the same test function:

PYTHON
```
import pytest

from meteor_watch.alerts import alert_level


@pytest.mark.parametrize(
    ("wind_kph", "visibility_km", "expected"),
    [
        (12, 10, "green"),
        (39.9, 10, "green"),
        (40, 10, "amber"),
        (69.9, 10, "amber"),
        (70, 10, "red"),
        (12, 4.9, "amber"),
        (12, 5, "green"),
        (12, 0.9, "red"),
        (12, 1, "amber"),
    ],
)
def test_alert_level_boundaries(wind_kph, visibility_km, expected):
    assert alert_level(wind_kph, visibility_km) == expected
```

Pytest collects nine cases, not one test containing nine hidden steps. Run:

BASH
```
python -m pytest --collect-only -q tests/test_alerts.py
```

Without explicit IDs, pytest creates IDs from values. They may be adequate for simple rows, but `test_alert_level_boundaries[12-10-green]` does not say why those numbers were chosen.

Pass an `ids` list aligned with the rows:

PYTHON
```
@pytest.mark.parametrize(
    ("wind_kph", "visibility_km", "expected"),
    [
        (12, 10, "green"),
        (39.9, 10, "green"),
        (40, 10, "amber"),
        (69.9, 10, "amber"),
        (70, 10, "red"),
    ],
    ids=[
```

PYTHON (CONTINUED)

```
        "calm-clear",
        "wind-just-below-amber",
        "wind-forty-is-amber",
        "wind-just-below-red",
        "wind-seventy-is-red",
    ],
)
def test_alert_level_wind_boundaries(wind_kph, visibility_km, expected):
    assert alert_level(wind_kph, visibility_km) == expected
```

Now a failure node can read:

TEXT

```
tests/test_alerts.py::test_alert_level_wind_boundaries[wind-seventy-is-red]
```

That node is directly selectable:

BASH

```
python -m pytest -q "tests/test_alerts.py::test_alert_level_wind_boundaries[wind-seventy-is-red]"
```

Shell quoting prevents brackets from being treated specially by some shells.

Put the ID beside a complicated row

`pytest.param` keeps the explanation adjacent to its data:

PYTHON

```
@pytest.mark.parametrize(
    ("wind_kph", "visibility_km", "expected"),
    [
        pytest.param(12, 10, "green", id="calm-clear"),
        pytest.param(40, 10, "amber", id="wind-forty-is-amber"),
        pytest.param(70, 10, "red", id="wind-seventy-is-red"),
        pytest.param(12, 1, "amber", id="visibility-one-is-amber"),
        pytest.param(12, 0.9, "red", id="visibility-below-one-is-red"),
    ],
)
def test_alert_level_selected_rules(wind_kph, visibility_km, expected):
    assert alert_level(wind_kph, visibility_km) == expected
```

Choose one ID style and keep it consistent within a table. IDs should explain the scenario, not repeat every literal.

Checkpoint: read a parametrized node

Turn contract rows into readable nodes

Choose the table or node that preserves useful failure evidence.

1. Which set best tests assignment of the 70-kph wind boundary?

- **A.** 10, 20, 30
- **B.** 70, 80, 90
- **C.** 69.9, 70, 70.1
- **D.** 39.9, 40, 40.1

2. Which parameter ID gives the clearest evidence for the row (70, 10, "red")?

- **A.** case-5
- **B.** 70-10-red
- **C.** test-data
- **D.** wind-seventy-is-red

3. A parametrized function has five rows. How many case nodes should pytest normally collect for it?

- **A.** Five separate nodes
- **B.** One node with five invisible results
- **C.** Six nodes because the function is also counted
- **D.** Zero until the function is called manually

Write your choices before checking the answer key at the end of this book.

3. Keep expected values independent and visible

This table looks compact but computes expected values using the same thresholds as production:

```python
wind_cases = [12, 40, 70]


def expected_level(wind_kph):
    if wind_kph < 40:
        return "green"
    if wind_kph < 70:
        return "amber"
    return "red"
```

If both functions accidentally use 71 as the red threshold, the tests agree with each other and disagree with the contract. Put small expected values directly in the rows. An independent reference model can be valuable for a complex algorithm, but then the model should be obviously simpler and tested or reviewed separately.

Explicit rows also make changed requirements reviewable:

```python
wind_cases = [
    pytest.param(39.9, "green", id="below-amber"),
    pytest.param(40, "amber", id="forty-is-amber"),
    pytest.param(69.9, "amber", id="below-red"),
    pytest.param(70, "red", id="seventy-is-red"),
]
```

If red risk later begins at 65, a reviewer can see which contract rows changed. Do not change production and every expected row before observing red. Update the test contract first, confirm the relevant case fails for the intended old behavior, then implement the new rule.

4. Prefer parametrization to a loop that hides the case

This test stops at the first failing iteration but reports only one node:

```python
def test_wind_cases_in_a_loop():
    cases = [
        (39.9, "green"),
        (40, "amber"),
        (70, "red"),
    ]
    for wind_kph, expected in cases:
        assert alert_level(wind_kph, 10) == expected
```

Pytest can show the local values, but you cannot select only the `70` row by node ID, and a failure at `40` prevents the later case from running. A loop is appropriate when iteration itself is the behavior under test. Use parametrization when the loop is merely a test-data delivery mechanism.

Do not compensate with a vague message:

```python
def test_wind_cases_with_a_message():
    for wind_kph, expected in [(40, "amber"), (70, "red")]:
        observed = alert_level(wind_kph, 10)
        assert observed == expected, f"failed for {wind_kph}"
```

The message helps, but separate parameter nodes still provide selection, collection counts, and independent outcomes.

5. Separate result and exception tables when they tell different stories

Valid cases return a label. Invalid measurements raise a specific exception. Trying to represent both with `expected` values and branches inside one test makes the assertion path harder to read:

```python
@pytest.mark.parametrize(
    ("wind_kph", "visibility_km", "expected_message"),
    [
        (-0.1, 10, "wind must be non-negative"),
        (10, -0.1, "visibility must be non-negative"),
    ],
)
def test_invalid_measurements(wind_kph, visibility_km, expected_message):
    with pytest.raises(ValueError, match=expected_message):
        alert_level(wind_kph, visibility_km)
```

The result table can remain a direct equality assertion. The invalid table can remain a direct exception assertion. The separation is useful because each group has one readable shape.

If exception types vary meaningfully, add `expected_exception` as data:

```python
@pytest.mark.parametrize(
    ("wind_kph", "expected_exception"),
    [
        pytest.param(-1, ValueError, id="negative-value"),
        pytest.param("fast", TypeError, id="non-numeric-value"),
    ],
)
def test_invalid_wind_types(wind_kph, expected_exception):
    with pytest.raises(expected_exception):
        validate_wind(wind_kph)
```

Only use this form when the production contract really distinguishes those types. Do not create a complicated universal table merely to avoid two clear test functions.

Checkpoint: keep each case understandable

Keep case data and assertions clear

Choose the design that preserves an independent oracle and useful failure.

1. Why is recalculating expected with the same threshold code as alert_level risky?

- **A.** Pytest cannot call helper functions
- **B.** The test oracle can repeat the same defect as production
- **C.** Expected values must always come from files
- **D.** Parametrization supports only strings

2. What important feature does parametrization provide that a loop inside one test does not?

- **A.** It makes production functions asynchronous
- **B.** It removes every repeated value
- **C.** A separately reported and selectable node for each row
- **D.** It guarantees every expected value is correct

3. Why can valid-result and invalid-exception cases use separate parametrized tests?

- **A.** Pytest prohibits two decorators in one file
- **B.** Exceptions cannot have IDs
- **C.** Every file may contain only one assertion type
- **D.** Each table then has one direct, easy-to-read assertion shape

Write your choices before checking the answer key at the end of this book.

6. Choose interacting cases without building the entire Cartesian product

Two inputs create combinations. Nine wind cases multiplied by nine visibility cases would produce 81 nodes. More cases are not automatically better if most repeat the same rule and hide the important interactions.

Use three layers:

1. isolate each threshold while the other input is safely ordinary;
2. add combinations where rules compete; and
3. add a named regression when a real interaction failed.

For example:

```python
@pytest.mark.parametrize(
    ("wind_kph", "visibility_km", "expected"),
```

PYTHON (CONTINUED)

```
    [
        pytest.param(70, 10, "red", id="red-from-wind"),
        pytest.param(12, 0.9, "red", id="red-from-visibility"),
        pytest.param(70, 0.9, "red", id="both-red-rules"),
        pytest.param(45, 0.9, "red", id="red-outranks-amber"),
        pytest.param(12, 4, "amber", id="amber-from-visibility"),
        pytest.param(45, 10, "amber", id="amber-from-wind"),
    ],
)
def test_alert_level_competing_rules(wind_kph, visibility_km, expected):
    assert alert_level(wind_kph, visibility_km) == expected
```

Each row adds a reason. `red-outranks-amber` checks rule priority. A dozen other red-plus-amber number pairs would not add the same amount of confidence.

Nested parametrization intentionally creates a Cartesian product:

PYTHON

```
@pytest.mark.parametrize("wind_kph", [10, 50, 80])
@pytest.mark.parametrize("visibility_km", [0.5, 3, 10])
def test_every_selected_weather_pair_returns_a_known_label(
    wind_kph,
    visibility_km,
):
    assert alert_level(wind_kph, visibility_km) in {
        "green",
        "amber",
        "red",
    }
```

This produces nine nodes and protects a broad vocabulary invariant. It does not replace exact expected results for priority and boundary rules. Use a Cartesian product when each combination is meaningful or the set is intentionally small.

7. Prevent parameter mutation from leaking between cases

Pytest passes parameter values as supplied; it does not copy mutable lists or dictionaries for each use. This production function mutates its argument:

PYTHON

```
def add_risk(record):
    record["risk"] = alert_level(
        record["wind_kph"],
        record["visibility_km"],
    )
    return record
```

Reusing the same dictionary object in more than one row can let the first case change the arrangement for the next. Prefer independent literals:

PYTHON

```
@pytest.mark.parametrize(
    ("record", "expected_risk"),
    [
        ({"wind_kph": 10, "visibility_km": 10}, "green"),
        ({"wind_kph": 70, "visibility_km": 10}, "red"),
    ],
)
def test_add_risk(record, expected_risk):
    observed = add_risk(record)
```

PYTHON (CONTINUED)

```
    assert observed["risk"] == expected_risk
```

If the same base object must generate variations, construct a fresh copy in the test or use a fixture factory in the next lesson. Do not use `deepcopy` by habit; first decide which nested objects the function is allowed to mutate.

8. Keep a discovered bug visible as a named regression

Suppose `alert_level(45, 0.9)` once returned `"amber"` because wind was checked before severe visibility. It could be another table row, but a dedicated test name preserves why the case matters:

PYTHON

```
def test_red_visibility_outranks_amber_wind_regression():
    assert alert_level(45, 0.9) == "red"
```

The separate name helps maintainers connect a future failure to the past bug. Use that visibility sparingly. If every ordinary row becomes a standalone “regression,” the suite loses its case-table structure.

9. Test the changed requirement before changing production

Imagine the wind red threshold changes from 70 to 65. Make this controlled change:

1. replace the relevant table IDs and values so `64.9` is amber and `65` is red;
2. run the selected `wind-sixty-five-is-red` case against the old production code;
3. confirm it fails with amber versus red;
4. change the production threshold once;
5. run all wind rows and competing-rule rows; and
6. inspect whether any former `69.9` expected value must change because the contract changed.

This order separates a requirement migration from an assertion edited only to make a current implementation pass.

Checkpoint: control combinations and shared data

Select interactions without creating noise

Use risk, independence, and the written contract to choose cases.

1. Which added row best checks that a red visibility rule outranks an amber wind rule?

- **A.** wind=45, visibility=0.9, expected="red"
- **B.** wind=45, visibility=10, expected="amber"
- **C.** wind=10, visibility=10, expected="green"
- **D.** wind=80, visibility=0.5, expected="red"

2. Why can reusing one dictionary object across parameter rows cause unreliable tests?

- **A.** Dictionaries cannot be function arguments

- **B.** A case may mutate it and change the arrangement seen by another case
- **C.** Pytest always converts it to a tuple
- **D.** Dictionary equality is approximate

3. When is a standalone named regression test especially useful?

- **A.** For every row in every table
- **B.** When no contract exists
- **C.** When its name preserves the meaning of a real past failure
- **D.** When it avoids asserting an expected result

Write your choices before checking the answer key at the end of this book.

10. Map every storm-warning boundary

Build `tests/test_alerts.py` from this contract:

- negative wind or visibility raises `ValueError` naming the invalid field;
- wind at least 70 or visibility below 1 produces `"red"`;
- otherwise, wind at least 40 or visibility below 5 produces `"amber"`;
- all other valid measurements produce `"green"`; and
- red conditions outrank amber conditions.

Your final suite should contain:

1. a wind-boundary table with meaningful IDs;
2. a visibility-boundary table with meaningful IDs;
3. a compact interaction table showing red outranks amber;
4. a separate invalid-input table using `pytest.raises`;
5. one named regression for `wind=45, visibility=0.9`;
6. no expected-value helper that repeats the production branches; and
7. no loop used merely to deliver cases.

Temporarily introduce two defects—`wind_kph > 70` and checking amber before red. Use the failed parameter IDs to identify each rule. Repair one defect at a time, run the focused node, then the whole file.

Hint 1: derive rows from each comparison

For wind, hold visibility at 10 and cover `39.9`, `40`, `69.9`, and `70`. For visibility, hold wind at 10 and cover `0.9`, `1`, `4.9`, and `5`.

Hint 2: make priority a competing pair

Use wind `45` to trigger amber and visibility `0.9` to trigger red. The combined expected result is red because the more severe condition wins.

Hint 3: keep invalid behavior in its own table

Use rows containing wind, visibility, and a stable message fragment. One test body can put only the `alert_level(...)` call inside `pytest.raises`.

Show a compact complete suite

PYTHON

```
import pytest

from meteor_watch.alerts import alert_level


@pytest.mark.parametrize(
    ("wind_kph", "expected"),
    [
        pytest.param(39.9, "green", id="below-amber"),
        pytest.param(40, "amber", id="forty-is-amber"),
        pytest.param(69.9, "amber", id="below-red"),
        pytest.param(70, "red", id="seventy-is-red"),
    ],
)
def test_wind_boundaries(wind_kph, expected):
    assert alert_level(wind_kph, 10) == expected


@pytest.mark.parametrize(
    ("visibility_km", "expected"),
    [
        pytest.param(0.9, "red", id="below-one-is-red"),
        pytest.param(1, "amber", id="one-is-amber"),
        pytest.param(4.9, "amber", id="below-five-is-amber"),
        pytest.param(5, "green", id="five-is-green"),
    ],
)
def test_visibility_boundaries(visibility_km, expected):
    assert alert_level(10, visibility_km) == expected


@pytest.mark.parametrize(
    ("wind_kph", "visibility_km", "expected"),
    [
        pytest.param(45, 0.9, "red", id="red-outranks-amber"),
        pytest.param(70, 4, "red", id="red-wind-outranks-amber"),
        pytest.param(45, 4, "amber", id="two-amber-rules"),
    ],
)
def test_competing_rules(wind_kph, visibility_km, expected):
    assert alert_level(wind_kph, visibility_km) == expected


@pytest.mark.parametrize(
    ("wind_kph", "visibility_km", "message"),
    [
        pytest.param(-0.1, 10, "wind", id="negative-wind"),
        pytest.param(10, -0.1, "visibility", id="negative-visibility"),
    ],
)
def test_negative_measurements(wind_kph, visibility_km, message):
    with pytest.raises(ValueError, match=message):
        alert_level(wind_kph, visibility_km)


def test_red_visibility_outranks_amber_wind_regression():
    assert alert_level(45, 0.9) == "red"
```

Key points

- Partition inputs by behavior before deciding which rows to write.

- Test just below, exactly at, and just above a meaningful boundary.
- Parametrization creates an independent, selectable node for every row.
- Give parameter cases IDs that explain why the input exists.
- Keep expected values explicit and independent of production decisions.
- Separate result, exception, and other differently shaped contracts when that keeps assertions direct.
- Add risk-driven interactions rather than an automatic huge Cartesian product.
- Do not share mutable parameter objects unless sharing is the behavior.
- Preserve a real past failure with a descriptive regression name when its history matters.

Continue learning

- pytest documentation: parametrizing tests
- pytest API reference: `pytest.param`
- pytest documentation: selecting tests by node ID and expression
- Next: Build an Isolated Test World with Fixtures

Build an Isolated Test World with Fixtures

Online lesson and runnable notebook

Use explicit pytest fixtures, fresh state, reliable teardown, temporary paths, fixture factories, captured streams, and captured logs to make failures reproducible and independent.

- **Level:** Python Foundations
- **Estimated time:** 4–5 hours
- **You will learn:** Construct an explicit fresh test world, compose and tear down fixtures, isolate file work with `tmp_path`, and assert terminal or log evidence without leaking state.
- **Practice in:** A local pytest project where fixture setup, teardown, temporary paths, capture, and logs can be observed

Meteor Watch can now classify observations. The next behavior writes an alert report. Testing it requires a directory, sample observations, and perhaps a captured message. If every test creates those resources by hand, important arrangement is repeated. If one global object is reused, mutations can make the suite order-dependent.

Pytest fixtures provide requested dependencies. A test names what it needs in its arguments; pytest constructs that test world, passes values in, and manages cleanup. Fixtures should make dependencies visible and isolation reliable—not hide the scenario that explains an expected result.

Keep these questions in view:

- Which setup belongs in a reusable fixture and which values should stay in the test?
- When does a fixture return a value, and when does it need teardown after `yield`?
- How do fixture dependencies determine setup and reverse teardown order?
- How do `tmp_path`, `capsys`, and `caplog` expose real boundaries safely?
- Which signs reveal shared state, an overly broad scope, or a fixture that owns too much behavior?

1. Request a fresh value by naming the fixture

Begin with a small dataclass and report function:

PYTHON

```
from dataclasses import dataclass
from pathlib import Path


@dataclass
class Observation:
    station: str
    wind_kph: float
    risk: str


def write_alert_report(path, observations):
    """Write one pipe-delimited alert per line and return the line count."""
    lines = [
```

PYTHON (CONTINUED)

```
        f"{item.station}|{item.wind_kph:g}|{item.risk}"
        for item in observations
    ]
    path.write_text("\n".join(lines) + "\n", encoding="utf-8")
    return len(lines)
```

A fixture is marked with `@pytest.fixture`:

PYTHON

```
import pytest

from meteor_watch.reports import Observation


@pytest.fixture
def observations():
    return [
        Observation("ridge-7", 72, "red"),
        Observation("lake-2", 18, "green"),
    ]
```

The test requests it by argument name:

PYTHON

```
def test_observations_fixture_supplies_two_records(observations):
    assert [item.station for item in observations] == [
        "ridge-7",
        "lake-2",
    ]
```

Pytest sees the argument, finds the fixture, calls it before the test, and passes its returned list. The fixture is not called manually. Calling `observations()` from a test bypasses pytest's dependency and lifecycle management.

The default scope is **function**: pytest calls the fixture separately for each test. This matters for mutable values:

PYTHON

```
def test_first_case_can_remove_a_record(observations):
    observations.pop()
    assert len(observations) == 1


def test_second_case_still_receives_two_records(observations):
    assert len(observations) == 2
```

Both tests pass in either order because each receives a newly constructed list. If `observations` were a module-level list returned repeatedly from a module-scoped fixture, the first mutation could leak.

Keep behavior-defining values in the test

This fixture hides the exact wind and expected label:

PYTHON

```
@pytest.fixture
def prepared_red_observation():
    return Observation("ridge-7", 72, "red")
```

It may be useful when many tests need the same complete object. But a threshold test is clearer with the boundary visible:

```python
def test_report_preserves_exact_red_boundary(tmp_path):
    observations = [Observation("ridge-7", 70, "red")]
    report_path = tmp_path / "alerts.txt"

    write_alert_report(report_path, observations)

    assert report_path.read_text(encoding="utf-8") == "ridge-7|70|red\n"
```

Use fixtures for environmental setup, costly construction, or shared shape. Keep the values that explain a test's expected behavior near the assertion.

2. Read the fixture dependency graph

Fixtures can request other fixtures:

```python
@pytest.fixture
def red_observation():
    return Observation("ridge-7", 72, "red")

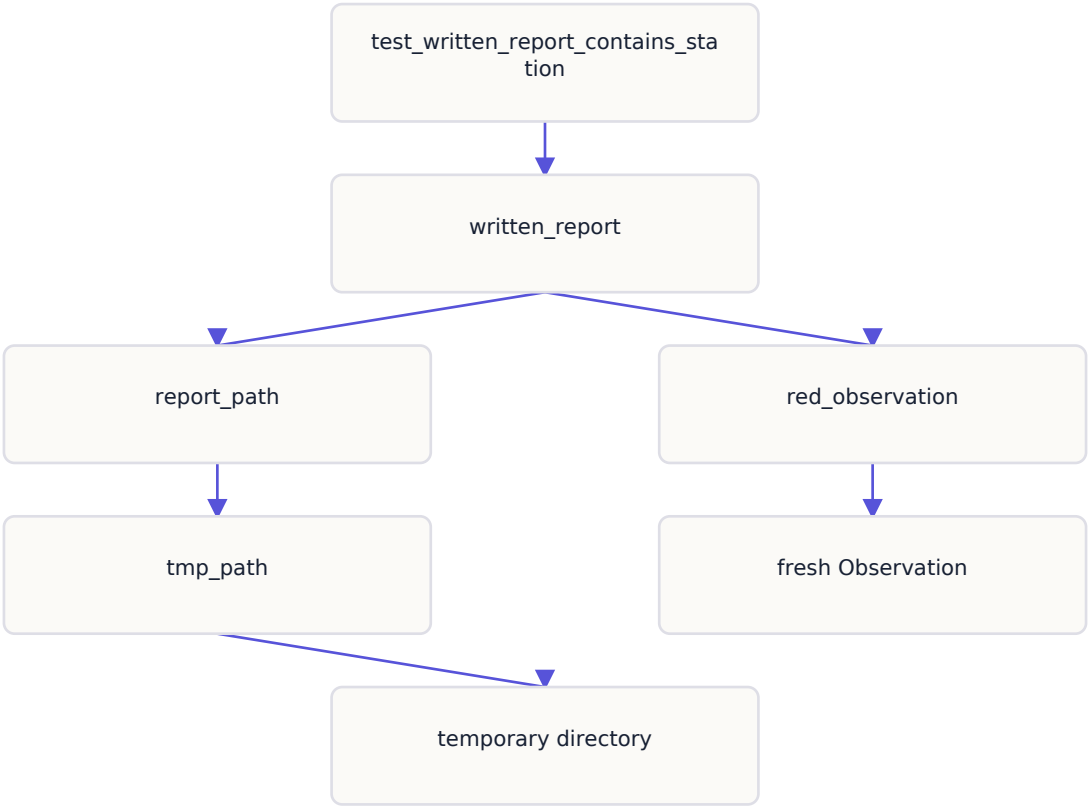

@pytest.fixture
def report_path(tmp_path):
    return tmp_path / "alerts.txt"


@pytest.fixture
def written_report(report_path, red_observation):
    write_alert_report(report_path, [red_observation])
    return report_path


def test_written_report_contains_station(written_report):
    text = written_report.read_text(encoding="utf-8")
    assert "ridge-7" in text
```

The test asks only for `written_report`. Pytest follows the dependency graph to `report_path`, `tmp_path`, and `red_observation`. Names describe resources; pytest determines a valid setup order from dependencies, scope, and request order rules. Do not rely on incidental definition order.

The requested fixture makes its dependencies visible. Teardown later proceeds in the reverse direction for fixtures that yielded resources.



Inspect the graph rather than guessing:

```bash
python -m pytest --setup-show -q tests/test_reports.py
```

The output labels fixture setup and teardown around each test. This is useful when state appears earlier than expected or cleanup fails.

Checkpoint: request only the world the test needs

Read fixture requests and fresh state

Choose the result that follows from pytest’s fixture lifecycle.

1. How does a test normally obtain the value from a fixture named observations?

- **A.** Import and call observations() manually
- **B.** Read a global named pytest
- **C.** Add @pytest.mark.observations
- **D.** Declare observations as a test-function argument

2. Two tests request a default function-scoped fixture that returns a new list. The first pops an item. What should the second normally receive?

- **A.** A newly constructed full list
- **B.** The shortened list from the first test
- **C.** No value because fixtures run once

- **D.** A tuple created by pytest

3. Where should a wind value chosen specifically to prove the 70-kph boundary usually appear?

- **A.** Hidden inside an unrelated session fixture
- **B.** Visible in the test or parameter row that asserts the boundary
- **C.** Only in conftest.py comments
- **D.** Inside pytest's cache

Write your choices before checking the answer key at the end of this book.

3. Use `yield` when the fixture owns cleanup

Some resources need an action after the test. Code before `yield` is setup; the yielded value goes to the test; code after `yield` is teardown:

```python
import os

import pytest


@pytest.fixture
def temporary_working_directory(tmp_path):
    original = Path.cwd()
    os.chdir(tmp_path)
    yield tmp_path
    os.chdir(original)
```

This example is educational, but pytest's `monkeypatch.chdir` is safer and will be introduced in Lesson 5. It automatically restores the original directory. The important structure is that a fixture which changes process state also owns its restoration.

For an object with explicit open/close behavior:

```python
class AlertChannel:
    def __init__(self):
        self.opened = False

    def open(self):
        self.opened = True

    def close(self):
        self.opened = False


@pytest.fixture
def alert_channel():
    channel = AlertChannel()
    channel.open()
    yield channel
    channel.close()
```

Teardown runs even if the test assertion fails after the fixture yields. It cannot undo setup that failed before `yield`, so keep risky state-changing actions small and pair each with the nearest cleanup. A context manager is often even clearer when the resource already supports one.

Prove teardown rather than assume it

Create an event list that a second fixture can inspect:

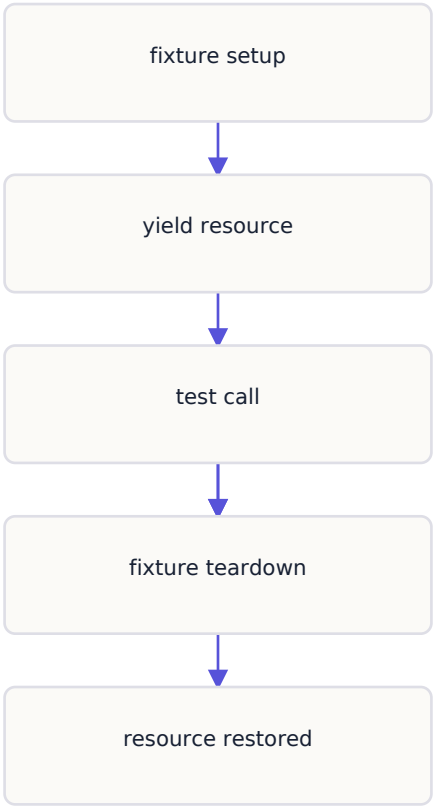
```python
@pytest.fixture
def events():
    return []


@pytest.fixture
def connected_channel(events):
    events.append("open")
    yield "channel"
    events.append("close")


def test_channel_is_open_during_call(connected_channel, events):
    assert connected_channel == "channel"
    assert events == ["open"]
```

To assert after teardown, a fixture or hook with a broader observation point is needed. For ordinary course code, `--setup-show` plus resource state after the test process is often enough. Do not contort production behavior merely to assert pytest's own tested lifecycle.

Teardown belongs to the fixture that performed setup and happens after the test call, even when the call is red.



test call → fixture teardown: pass

test call → fixture teardown: failure

4. Choose the narrowest useful fixture scope

Fixtures support `function`, `class`, `module`, `package`, and `session` scopes. Broader scope can reduce expensive repeated setup, but it also expands shared state and the distance between cause and failure.

```python
@pytest.fixture(scope="session")
def static_station_catalog():
    return (
        ("ridge-7", 1200),
        ("lake-2", 340),
    )
```

An immutable catalog is a reasonable session value. A mutable list of current alerts is not: one test can change what every later test sees. Start with function scope. Widen only after measurement shows setup cost matters and the resource has an explicit sharing/isolation contract.

Scope dependencies also move in one direction: a broader-scoped fixture cannot depend on a narrower function-scoped value whose lifetime ends sooner. If pytest reports `ScopeMismatch`, redesign the resource lifetimes rather than randomly changing scope strings.

5. Put shared fixtures near their consumers

Pytest discovers fixtures in test modules, parent `conftest.py` files, and plugins. A fixture used by tests in one file can stay in that file. A fixture shared by one test package can move to the nearest `tests/conftest.py`:

```text
tests/
├── conftest.py
├── test_alerts.py
└── test_reports.py
```

Example `conftest.py`:

```python
import pytest

from meteor_watch.reports import Observation


@pytest.fixture
def observation_factory():
    def build(
        station="ridge-7",
        wind_kph=72,
        risk="red",
    ):
        return Observation(station, wind_kph, risk)

    return build
```

A **fixture factory** supplies fresh variations without hiding the case:

```python
def test_report_preserves_station_order(tmp_path, observation_factory):
    observations = [
        observation_factory(station="ridge-7"),
```

PYTHON (CONTINUED)

```
        observation_factory(station="tower-1", wind_kph=45, risk="amber"),
    ]
    path = tmp_path / "alerts.txt"

    write_alert_report(path, observations)

    assert path.read_text(encoding="utf-8").splitlines() == [
        "ridge-7|72|red",
        "tower-1|45|amber",
    ]
```

Avoid a giant root `conftest.py` with unrelated application behavior. Hidden fixture availability makes it difficult to know where a value came from. Search should lead from the requesting test to the nearest owner.

Checkpoint: teardown, scope, and ownership

Choose safe fixture lifetimes

Use setup ownership, cleanup, and mutability to choose the fixture design.

1. In a yield fixture, when does code after yield normally run?

- **A.** During test collection
- **B.** Before dependency setup
- **C.** During teardown after the test call, including after an assertion failure
- **D.** Only when every test passes

2. Which resource is the safest candidate for session scope?

- **A.** A list every test appends to
- **B.** The current working directory
- **C.** One temporary output file shared by writers
- **D.** An immutable station catalog that is expensive to load

3. Where should a fixture used only by tests/test_reports.py normally begin?

- **A.** In test_reports.py near its consumers
- **B.** In the repository root conftest.py
- **C.** Inside production **init**.py
- **D.** In pytest’s installed source

Write your choices before checking the answer key at the end of this book.

6. Use `tmp_path` for a real isolated filesystem boundary

`tmp_path` is a built-in function-scoped fixture that supplies a unique `pathlib.Path` directory for each test. It lets production file code use real filesystem operations without writing into the repository or sharing output names.

Test a complete write/read round trip:

```python
def test_write_alert_report_round_trips_utf8(tmp_path):
    path = tmp_path / "alerts.txt"
    observations = [Observation("serra-ç", 72, "red")]

    count = write_alert_report(path, observations)
    observed_text = path.read_text(encoding="utf-8")

    assert count == 1
    assert observed_text == "serra-ç|72|red\n"
```

The test owns the directory, chooses the filename, uses the production writer, and reads actual UTF-8 bytes through `Path`. This is an integration with the local filesystem, but it remains fast and deterministic.

Prove paths are different

Do not assert an exact temporary directory name; it varies. Assert properties that matter:

```python
def test_tmp_path_is_an_empty_directory_for_this_test(tmp_path):
    assert tmp_path.is_dir()
    assert list(tmp_path.iterdir()) == []
```

A later test receives another path. Pytest retains several recent temporary roots to aid debugging, so tests should not assume the directory is deleted at the exact end of the call. Isolation—not immediate deletion—is the public fixture promise relevant here.

Use `tmp_path_factory` when a genuinely expensive immutable file should be created once for a broader scope. Most beginner file tests should start with `tmp_path` because its per-test isolation is easier to trust.

7. Capture stdout and stderr with `capsys`

Suppose the application prints a short completion message:

```python
def announce_report(count):
    noun = "alert" if count == 1 else "alerts"
    print(f"Wrote {count} {noun}")
```

The built-in `capsys` fixture captures writes to Python's `sys.stdout` and `sys.stderr`:

```python
def test_announce_report_uses_singular(capsys):
    announce_report(1)

    captured = capsys.readouterr()

    assert captured.out == "Wrote 1 alert\n"
    assert captured.err == ""
```

`readouterr()` returns the output captured so far and resets the buffers for the next interval:

```python
def test_two_announcements_can_be_checked_separately(capsys):
    announce_report(1)
    first = capsys.readouterr()
```

PYTHON (CONTINUED)

```
    announce_report(2)
    second = capsys.readouterr()

    assert first.out == "Wrote 1 alert\n"
    assert second.out == "Wrote 2 alerts\n"
```

`capsys` captures Python-level text streams. `capfd` captures operating-system file descriptors and can observe output from some subprocesses or extension code. Choose the narrowest fixture that matches the actual output boundary.

Pytest captures output by default and displays it with a failure. `capsys` is for asserting that output as public behavior or separating phases—not for printing more debug noise and hoping it appears.

8. Capture logging records with `caplog`

Logs are structured diagnostic records, not merely stderr text. Test their level and message through `caplog`:

PYTHON

```
import logging


logger = logging.getLogger("meteor_watch.reports")


def log_rejected_station(station, reason):
    logger.warning("rejected station=%s reason=%s", station, reason)


def test_rejected_station_emits_warning(caplog):
    with caplog.at_level(logging.WARNING, logger="meteor_watch.reports"):
        log_rejected_station("ridge-7", "missing wind")

    assert caplog.record_tuples == [
        (
            "meteor_watch.reports",
            logging.WARNING,
            "rejected station=ridge-7 reason=missing wind",
        )
    ]
```

Avoid asserting timestamps or default renderer punctuation unless the formatted log line itself is the external contract. Records expose stable fields. If the application removes or replaces handlers during a test, `caplog` may no longer see records; treat logging configuration as application-boundary behavior from Unit 12.

Checkpoint: choose the observable boundary

Isolate files, output, and logs

Choose the fixture that matches the evidence in the contract.

1. Why is tmp_path a strong default for testing a report writer?

- **A.** It mocks every Path method
- **B.** It supplies a unique real Path directory for each test
- **C.** It forces all tests to share one output

- **D.** It prevents the production function from opening files

2. After capsys.readouterr(), what do captured.out and captured.err represent?

- **A.** The source code of print
- **B.** Only logging records
- **C.** Captured Python stdout and stderr text so far
- **D.** The child process exit status

3. Why prefer caplog records over an exact default rendered log line?

- **A.** Records cannot include messages
- **B.** caplog disables levels
- **C.** Rendered logs are always empty
- **D.** Records expose stable logger, level, and message fields without incidental timestamp formatting

Write your choices before checking the answer key at the end of this book.

9. Diagnose a leaking test world

Order-dependent failures often appear only in the full suite. Look for:

- a module or class variable mutated by a test;
- a fixture whose scope is broader than its mutable value;
- a temporary path stored globally and reused;
- a working-directory or environment change without restoration;
- a fixture that returns the same object repeatedly;
- a cached production function whose cache is not part of the arrangement; or
- a test that calls another test instead of arranging its own state.

Use `--setup-show` to inspect lifecycle and run each failed node alone. If a node passes alone but fails after another node, run the two in that order and inspect what the first changes. Do not add sleeps or depend on a preferred order.

This intentionally shared fixture is dangerous:

PYTHON

```
@pytest.fixture(scope="module")
def shared_records():
    return []


def test_adds_red_record(shared_records):
    shared_records.append("red")
    assert shared_records == ["red"]


def test_starts_without_records(shared_records):
    assert shared_records == []
```

Change it to default function scope, or return an immutable value and let each test create its own list. The correct choice follows the application contract: if shared history is not what you are testing, do not share it.

10. Give every alert a clean workspace

Create a final lab around `write_alert_report`, `announce_report`, and `log_rejected_station`. Your suite must:

1. define an `observation_factory` fixture with useful defaults;
2. prove two tests receive fresh mutable observation lists;
3. write and read a UTF-8 report through `tmp_path`;
4. prove two separately requested temporary paths are not assumed to match;
5. capture exact singular and plural stdout without asserting unrelated output;
6. capture one warning's logger name, level, and message;
7. include one `yield` fixture for a small resource that records open/close events;
8. deliberately create the module-scoped list leak, reproduce it, and repair it by correcting ownership or scope; and
9. retain one `--setup-show` excerpt that explains setup and teardown order.

Run the report file alone, then the complete suite in a new process. A passing preferred order is not sufficient evidence.

Hint 1: let the factory vary only meaningful fields

Return an inner `build` function with defaults for station, wind, and risk. Each call should construct a new `Observation`; do not keep one instance in the outer fixture.

Hint 2: read captured streams immediately after the action

Call `captured = capsys.readouterr()` after each announcement. Assert `captured.out` and `captured.err` separately.

Hint 3: pair state-changing setup with its teardown

Append `"open"` before `yield` and `"close"` after it. Use `--setup-show` or a broader event observer to verify the sequence without making a later test depend on shared mutable history.

Show the central file and capture checks

PYTHON

```
import logging

import pytest

from meteor_watch.reports import (
    Observation,
    announce_report,
    log_rejected_station,
    write_alert_report,
)


@pytest.fixture
def observation_factory():
    def build(station="ridge-7", wind_kph=72, risk="red"):
```

PYTHON (CONTINUED)

```
        return Observation(station, wind_kph, risk)

    return build


def test_report_round_trips_utf8(tmp_path, observation_factory):
    path = tmp_path / "alerts.txt"
    records = [observation_factory(station="serra-ç")]

    count = write_alert_report(path, records)

    assert count == 1
    assert path.read_text(encoding="utf-8") == "serra-ç|72|red\n"


@pytest.mark.parametrize(
    ("count", "expected"),
    [(1, "Wrote 1 alert\n"), (2, "Wrote 2 alerts\n")],
)
def test_announcement_pluralization(count, expected, capsys):
    announce_report(count)
    captured = capsys.readouterr()

    assert captured.out == expected
    assert captured.err == ""


def test_rejection_is_a_warning_record(caplog):
    with caplog.at_level(logging.WARNING, logger="meteor_watch.reports"):
        log_rejected_station("ridge-7", "missing wind")

    assert caplog.record_tuples == [
        (
            "meteor_watch.reports",
            logging.WARNING,
            "rejected station=ridge-7 reason=missing wind",
        )
    ]
```

The full lab should add the fresh-list, different-path, lifecycle, and repaired state-leak evidence described above.

Key points

- A test requests fixtures through arguments; pytest owns resolution and lifecycle.
- Function scope gives mutable fixtures fresh state by default.
- Keep behavior-defining values visible in the test or parameter row.
- Fixtures may depend on fixtures, creating an explicit setup graph.
- A `yield` fixture owns teardown after the test call, including after a failed assertion.
- Widen scope only for a measured reason and a resource safe to share.
- Place fixtures near their consumers and use factories for fresh variations.
- `tmp_path` exercises a real isolated filesystem boundary.
- `capsys` captures stdout/stderr text; `caplog` captures structured log records.
- Reproduce order-dependent failures and repair state ownership rather than relying on execution order.

Continue learning

- pytest documentation: fixtures and dependency injection
- pytest documentation: temporary directories and files
- pytest documentation: stdout and stderr capture
- pytest documentation: capturing log records
- Next: Control Dependencies Without Fragile Mocks

Control Dependencies Without Fragile Mocks

Online lesson and runnable notebook

Make time, randomness, environment settings, and collaborators deterministic through explicit seams, small fakes, pytest monkeypatching, and narrowly specified mocks that protect public obligations.

- **Level:** Python Foundations
- **Estimated time:** 4–5 hours
- **You will learn:** Identify uncontrollable dependencies, introduce the smallest useful seam, patch state where code looks it up, and avoid interaction assertions that reject safe implementation changes.
- **Practice in:** A local pytest project using explicit collaborators, pytest's `monkeypatch`, and Python's standard-library `unittest.mock`

Meteor Watch should include a timestamp in each alert and send a notification for red risk. A direct implementation can reach the system clock, environment, random generator, and real notification service. The behavior may work in production while remaining slow, destructive, or impossible to reproduce in a test.

The answer is not "mock everything." First make important dependencies visible. Then choose the simplest controlled replacement that provides the evidence you need. A returned result may need only a stub value. A stateful in-memory repository may deserve a fake. A public send-once obligation may justify a mock interaction assertion.

Use these questions throughout:

- Which hidden dependency makes the result nondeterministic or destructive?
- Can an explicit parameter or small collaborator create a natural seam?
- Is the contract about returned state, visible output, or an interaction?
- Where does the system under test look up the name that must be patched?
- Would this assertion still pass after a safe internal refactor?

1. Expose the clock instead of racing it

This function hides the current time:

```python
from datetime import datetime, timezone


def build_alert(station, risk):
    created_at = datetime.now(timezone.utc)
    return {
        "station": station,
        "risk": risk,
        "created_at": created_at.isoformat(),
    }
```

A test cannot know the exact microsecond in advance. Asserting “the timestamp is near now” adds timing tolerance and can still fail on a busy machine. Sleeping does not make the dependency deterministic.

Inject a callable with the behavior the function needs:

```python
from datetime import datetime, timezone


def utc_now():
    return datetime.now(timezone.utc)


def build_alert(station, risk, clock=utc_now):
    created_at = clock()
    return {
        "station": station,
        "risk": risk,
        "created_at": created_at.isoformat(),
    }
```

The ordinary caller uses the real default. The test supplies a deterministic clock:

```python
from datetime import datetime, timezone


def fixed_clock():
    return datetime(2035, 6, 1, 12, 30, tzinfo=timezone.utc)


def test_build_alert_uses_supplied_clock():
    observed = build_alert("ridge-7", "red", clock=fixed_clock)

    assert observed == {
        "station": "ridge-7",
        "risk": "red",
        "created_at": "2035-06-01T12:30:00+00:00",
    }
```

The injected dependency is small: “a callable returning an aware datetime.” It does not expose an entire framework or require the test to alter process-global time.

Control randomness through the operation you need

Suppose equal-risk alerts are assigned a rotating radio channel:

```python
import random


def choose_channel(channels, chooser=random.choice):
    if not channels:
        raise ValueError("at least one channel is required")
    return chooser(channels)
```

A deterministic chooser can return the last item:

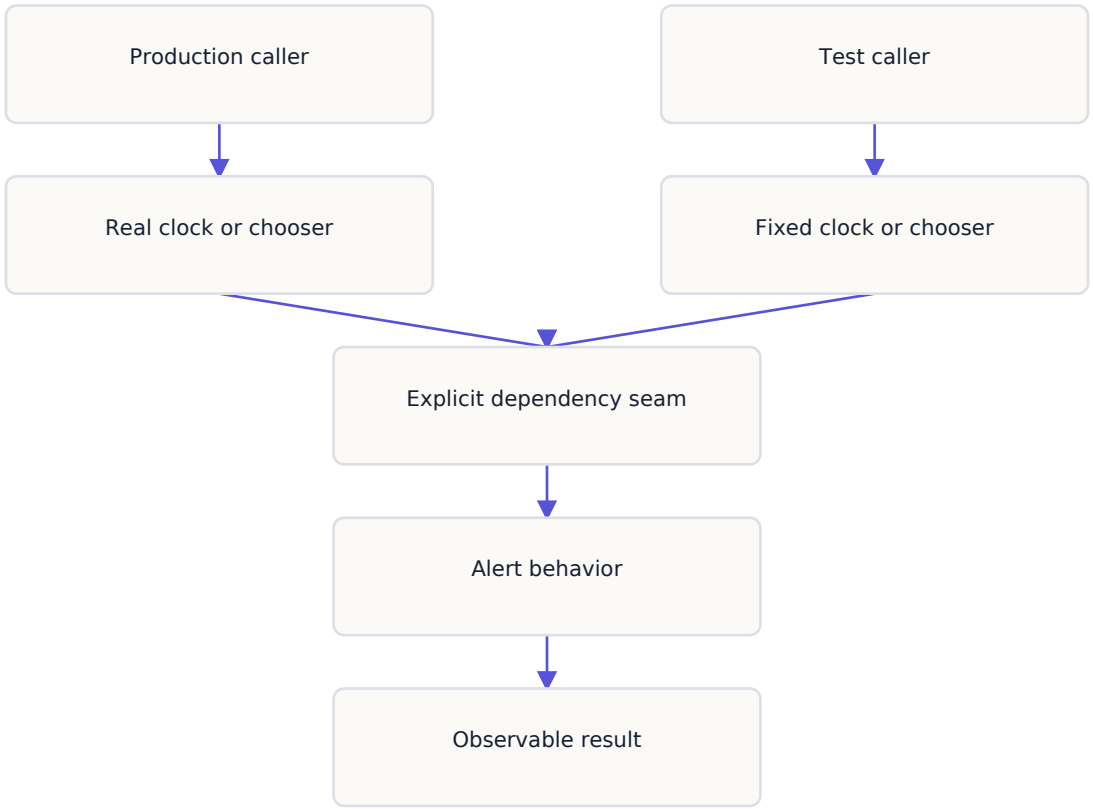
```python
def choose_last(values):
    return values[-1]
```

PYTHON (CONTINUED)

```
def test_choose_channel_uses_supplied_chooser():
    assert choose_channel(["alpha", "beta"], chooser=choose_last) == "beta"
```

This proves delegation and result behavior without globally seeding randomness. Use a seeded real generator only when the sequence produced by that generator is itself the subject. Tests should not depend on CPython’s exact random sequence unless that is an intentional compatibility contract.

The seam lets production and test callers supply different collaborators while the core behavior remains the same.



2. Choose a double by the evidence it supplies

Test-double vocabulary is useful when it communicates behavior:

Kind	Supplies	Meteor Watch use
dummy	required shape but no meaningful behavior	an unused context argument
stub	a prepared answer	fixed clock returning one datetime
fake	working lightweight implementation	in-memory notification outbox
spy	records calls for later inspection	sender that retains delivered alerts

Kind	Supplies	Meteor Watch use
mock	preconfigured interaction expectations	spec-constrained sender checked for one public call

Tools and teams use these words with small variations. The design question is more important: what controlled behavior and observation does the test need?

An in-memory fake can be ordinary Python:

```python
class MemorySender:
    def __init__(self):
        self.sent = []

    def send(self, alert):
        self.sent.append(alert)


def notify_if_red(alert, sender):
    if alert["risk"] == "red":
        sender.send(alert)
        return True
    return False


def test_red_alert_is_delivered_to_memory_sender():
    sender = MemorySender()
    alert = {"station": "ridge-7", "risk": "red"}

    delivered = notify_if_red(alert, sender)

    assert delivered is True
    assert sender.sent == [alert]
```

This asserts public result and final observable state. The fake has no network, sleep, credentials, or retry policy. It is easy to read and can support several tests without a mock API.

Checkpoint: choose the smallest controllable seam

Control time, randomness, and sending

Choose the dependency design that produces deterministic evidence.

1. What is the strongest reason to pass a clock callable to build_alert?

- **A.** The test can supply one exact time without racing the real clock
- **B.** Datetime objects cannot be asserted
- **C.** Pytest refuses to import datetime
- **D.** Production code must never use the real clock

2. A MemorySender implements send() and stores delivered alerts. Which description fits best?

- **A.** A real remote service
- **B.** A lightweight fake or spy with observable state
- **C.** A pytest collection hook
- **D.** A coverage report

3. Which test design makes choose_channel deterministic without changing global random state?

- **A.** Sleep before calling it
- **B.** Assert any string
- **C.** Pass a chooser callable with a prepared result
- **D.** Retry until the expected channel appears

Write your choices before checking the answer key at the end of this book.

3. Patch process state temporarily with `monkeypatch`

Some application boundaries intentionally read environment variables or the current directory. Redesigning every read as a parameter can make a thin adapter awkward. Pytest's `monkeypatch` fixture changes state for one test and restores it afterward.

Example production setting:

```python
import os


def alert_region():
    return os.environ.get("METEOR_REGION", "local").strip().casefold()
```

Test an explicit value:

```python
def test_alert_region_reads_environment(monkeypatch):
    monkeypatch.setenv("METEOR_REGION", " NORTH ")

    assert alert_region() == "north"
```

Test absence separately:

```python
def test_alert_region_defaults_when_variable_is_absent(monkeypatch):
    monkeypatch.delenv("METEOR_REGION", raising=False)

    assert alert_region() == "local"
```

After each test, pytest restores the previous environment. Do not manually put a guessed old value back; the variable may originally have been absent or held a user-specific value.

`monkeypatch` can also:

- `setitem` or `delitem` on mappings;
- `setattr` or `delattr` on objects/modules;
- prepend to `sys.path` for specific import tests;
- `chdir` temporarily; and
- create a limited `context()` for a smaller patch lifetime.

Use these operations at the boundary that deliberately owns process state. A pure classifier should accept a value rather than read the environment in every call.

Change directories without leaking the process state

```python
from pathlib import Path


def current_station_file():
    return Path("stations.json")


def test_station_file_is_relative_to_selected_workspace(
    tmp_path,
    monkeypatch,
):
    monkeypatch.chdir(tmp_path)

    assert current_station_file().resolve() == tmp_path / "stations.json"
```

The test makes the current-directory dependency explicit and restored. An even more reusable design would accept a base path, but a CLI adapter whose contract uses the current directory may intentionally be tested this way.

4. Patch the name the module actually looks up

Patch failures often come from changing the name where a function was defined instead of where the system under test imported it.

Suppose `meteor_watch/alerts.py` contains:

```python
from meteor_watch.delivery import send_alert


def publish(alert):
    send_alert(alert)
```

At import time, `alerts` binds its own name `send_alert`. Patching `meteor_watch.delivery.send_alert` later does not replace the already-bound name used by `publish`. Patch `meteor_watch.alerts.send_alert`:

```python
import meteor_watch.alerts


def test_publish_uses_alert_module_sender(monkeypatch):
    sent = []

    def fake_send(alert):
        sent.append(alert)

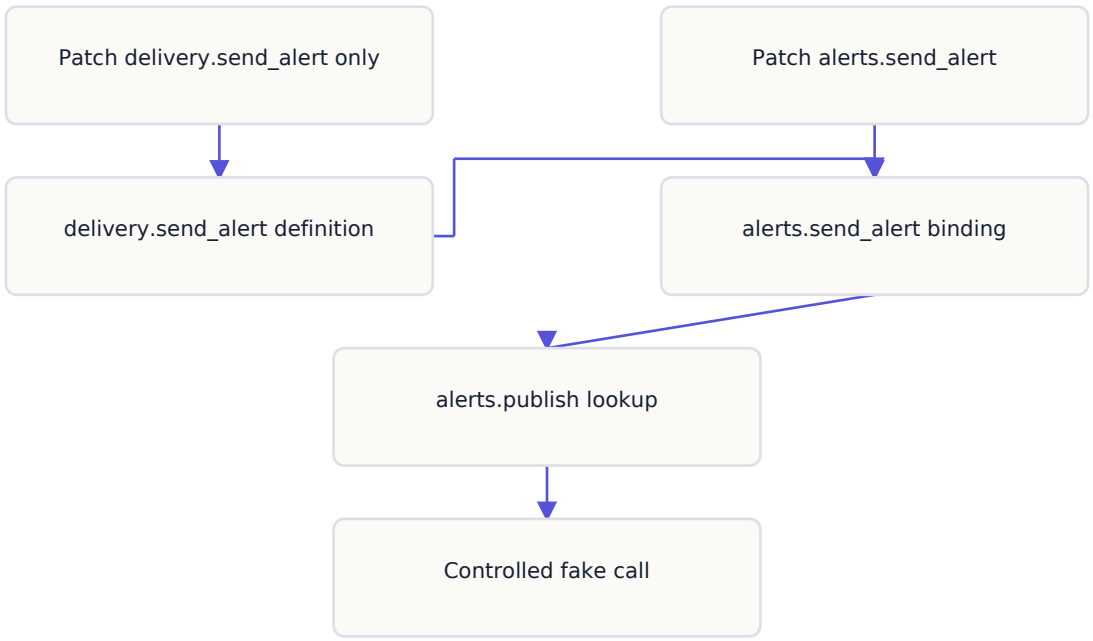
    monkeypatch.setattr(meteor_watch.alerts, "send_alert", fake_send)
    alert = {"station": "ridge-7", "risk": "red"}

    meteor_watch.alerts.publish(alert)

    assert sent == [alert]
```

If production instead used `import meteor_watch.delivery` and called `meteor_watch.delivery.send_alert(...)`, that is the name path it looks up and the patch location changes accordingly.

The caller follows its local binding. A patch must replace that lookup path, not merely another module’s original definition.



Patch delivery.send_alert only → delivery.send_alert definition: does not replace bound name

Reproduce the wrong-namespace symptom

Patch the definition module on purpose and run the focused test. If the real sender raises `RuntimeError("network disabled")`, seeing that exception proves the fake was not used. Inspect the import statement in the system under test, patch its lookup name, and rerun. This is stronger than adding another patch at random.

5. Use `Mock` when the interaction is the behavior

Python’s `unittest.mock` can configure return values, raise side effects, and record calls. Constrain the mock to a known collaborator shape:

```python
from unittest.mock import Mock


class Sender:
    def send(self, alert):
        raise NotImplementedError


def test_red_alert_is_sent_once():
    sender = Mock(spec=Sender)
    alert = {"station": "ridge-7", "risk": "red"}

    delivered = notify_if_red(alert, sender)

    assert delivered is True
    sender.send.assert_called_once_with(alert)
```

`spec=Sender` prevents a test from configuring imaginary attributes that the real interface lacks. `create_autospec(Sender, instance=True)` can also enforce method signatures. A spec does not prove the real service works; it catches some test-double drift.

Use `return_value` when production consumes a collaborator's result:

```python
from unittest.mock import Mock


def next_sequence(counter):
    return counter.next_value()


def test_next_sequence_returns_counter_value():
    counter = Mock()
    counter.next_value.return_value = 17

    assert next_sequence(counter) == 17
```

Use `side_effect` to model an anticipated collaborator failure:

```python
from unittest.mock import Mock


def safe_send(alert, sender):
    try:
        sender.send(alert)
    except ConnectionError:
        return False
    return True


def test_safe_send_reports_connection_failure():
    sender = Mock(spec=Sender)
    sender.send.side_effect = ConnectionError("radio offline")

    assert safe_send({"risk": "red"}, sender) is False
```

Do not use a broad side effect merely to make every failure path convenient. The production contract should name which collaborator failures it handles.

Checkpoint: patch and specify the real seam

Patch state and collaborator lookups safely

Choose the patch or mock behavior that matches the code's lookup and contract.

1. Why use monkeypatch.setenv instead of manually assigning os.environ and guessing a cleanup value?

- **A.** It makes environment variables immutable
- **B.** It permanently changes every process
- **C.** It disables pytest capture
- **D.** Pytest restores the actual prior state after the test

2. alerts.py uses "from delivery import send_alert" and publish() calls its bound send_alert name. Where should the test patch?

- **A.** The send_alert name in the alerts module
- **B.** Every send function in the interpreter
- **C.** The test function’s local namespace
- **D.** Only the source text on disk

3. What benefit does Mock(spec=Sender) provide?

- **A.** It contacts the real sender
- **B.** It rejects access to attributes outside the known Sender interface
- **C.** It guarantees all call assertions are good design
- **D.** It makes network tests end-to-end

Write your choices before checking the answer key at the end of this book.

6. Prefer observable state unless interaction is a public obligation

This test is brittle:

```
PYTHON
def test_report_builder_calls_three_helpers(mocker):
    parse = mocker.patch("meteor_watch.reports.parse")
    classify = mocker.patch("meteor_watch.reports.classify")
    render = mocker.patch("meteor_watch.reports.render")

    build_report("ridge-7|70")

    parse.assert_called_once()
    classify.assert_called_once()
    render.assert_called_once()
```

It also requires the third-party `pytest-mock` plugin, which this course does not install. More importantly, the assertions freeze three private steps. A refactor that combines parsing and classification breaks the test even if the returned report is identical.

Protect public output instead:

```
PYTHON
def test_build_report_returns_classified_station():
    assert build_report("ridge-7|70") == "ridge-7|red"
```

An interaction assertion is appropriate when interaction **is** the promise:

- send a red alert exactly once;
- do not send a green alert;
- commit a transaction only after all records validate; or
- release a resource acquired by the function.

Even then, assert the narrow obligation, not every internal call:

```python
from unittest.mock import Mock


def test_green_alert_is_not_sent():
    sender = Mock(spec=Sender)
    alert = {"station": "lake-2", "risk": "green"}

    delivered = notify_if_red(alert, sender)

    assert delivered is False
    sender.send.assert_not_called()
```

7. Avoid mock chains and imaginary worlds

This arrangement is hard to relate to a real interface:

```python
client.session.return_value.channel.return_value.send.return_value = {
    "ok": True
}
```

Deep chains reproduce implementation navigation and allow a test-only world that no real client supports. Hide a complex vendor client behind a small application-owned adapter such as `Sender.send(alert)`. Test core code against that small interface, and give the adapter a few integration tests at its real boundary when appropriate.

Do not patch Python builtins such as `open` across the whole process when `tmp_path` can exercise a real file. Do not patch `time.sleep` as a substitute for a design with injectable retry policy. Do not send to a real production service from a normal unit test.

8. Compare dependency injection with patching

Both techniques can be valid:

Situation	Prefer
core behavior already accepts a collaborator	pass a stub, fake, or mock explicitly
a new design naturally benefits from visible dependencies	dependency injection
thin adapter intentionally reads environment/current directory	`monkeypatch` that state
legacy module bound an imported function	patch the lookup namespace temporarily
real local filesystem is fast and deterministic	`tmp_path`, not a fake file API
interaction is not public behavior	assert returned state/output instead of calls

Injection improves design visibility. Patching changes an existing lookup for a limited test lifetime. Patching every dependency can conceal a tightly coupled design; forcing every process adapter to accept dozens of parameters can also reduce clarity. Choose the smallest seam at the layer that owns the boundary.

Checkpoint: reject fragile interaction tests

Protect obligations without freezing internals

Choose the assertion or seam that survives acceptable design changes.

1. The contract promises build_report returns "ridge-7|red" but says nothing about helpers. Which check is most stable?

- **A.** Assert parse, classify, and render are each called once
- **B.** Patch every private function
- **C.** Assert the returned report equals "ridge-7|red"
- **D.** Assert the implementation has three functions

2. When is assert_called_once_with(alert) most defensible?

- **A.** Whenever any helper exists
- **B.** When it increases line coverage
- **C.** When no result can be observed
- **D.** When sending exactly one alert is part of the public obligation

3. A report writer should create UTF-8 text in a local file. Which test boundary is the clearest default?

- **A.** Use a real isolated path from tmp_path
- **B.** Patch every open and Path method
- **C.** Write into the repository
- **D.** Skip the file behavior

Write your choices before checking the answer key at the end of this book.

9. Make the meteor alert deterministic

Build a small alert workflow with these public functions:

```python
def build_alert(station, risk, clock):
    """Return an alert dictionary stamped with clock()."""
    return {
        "station": station,
        "risk": risk,
        "created_at": clock().isoformat(),
    }


def publish_if_enabled(alert, sender, enabled):
    """Send red alerts when enabled and report whether delivery occurred."""
    if not enabled or alert["risk"] != "red":
        return False
    sender.send(alert)
    return True
```

Add a thin environment adapter:

```python
import os


def notifications_enabled():
    raw = os.environ.get("METEOR_NOTIFY", "0").strip().casefold()
    return raw in {"1", "true", "yes"}
```

Your tests must:

1. inject a fixed aware datetime and assert the complete alert;
2. use a `MemorySender` to prove red delivery and green non-delivery through final state;
3. use `monkeypatch.setenv` and `delenv` for enabled, disabled, and absent environment cases;
4. use `Mock(spec=Sender)` for exactly one test where send-once is the public obligation;
5. configure `side_effect=ConnectionError(...)` for an explicitly handled sender failure;
6. reproduce one wrong-namespace patch and record the real evidence;
7. replace one assertion on private helper calls with a returned-state or output assertion; and
8. run every test both alone and as part of the complete file.

Hint 1: make a clock no more complicated than necessary

A zero-argument function returning one timezone-aware `datetime` is sufficient. The workflow needs `clock()`, not a fake calendar library.

Hint 2: decide whether state or interaction is the evidence

Use `MemorySender.sent` for most tests. Reserve `assert_called_once_with` for the explicit exactly-once delivery obligation.

Hint 3: follow the import statement to the patch target

If `workflow.py` imported `send_alert` directly, patch `meteor_watch.workflow.send_alert`. If it imported the module and accesses an attribute, patch that attribute on the module object it uses.

Show a deterministic core suite

```python
from datetime import datetime, timezone
from unittest.mock import Mock


class Sender:
    def send(self, alert):
        raise NotImplementedError


class MemorySender:
    def __init__(self):
        self.sent = []

    def send(self, alert):
        self.sent.append(alert)


def fixed_clock():
    return datetime(2035, 6, 1, 12, 30, tzinfo=timezone.utc)
```

PYTHON (CONTINUED)

```
def test_build_alert_uses_injected_clock():
    assert build_alert("ridge-7", "red", fixed_clock) == {
        "station": "ridge-7",
        "risk": "red",
        "created_at": "2035-06-01T12:30:00+00:00",
    }


def test_red_alert_reaches_memory_sender():
    sender = MemorySender()
    alert = build_alert("ridge-7", "red", fixed_clock)

    assert publish_if_enabled(alert, sender, enabled=True) is True
    assert sender.sent == [alert]


def test_green_alert_is_not_delivered():
    sender = MemorySender()
    alert = build_alert("lake-2", "green", fixed_clock)

    assert publish_if_enabled(alert, sender, enabled=True) is False
    assert sender.sent == []


def test_red_alert_is_sent_exactly_once():
    sender = Mock(spec=Sender)
    alert = build_alert("ridge-7", "red", fixed_clock)

    delivered = publish_if_enabled(alert, sender, enabled=True)

    assert delivered is True
    sender.send.assert_called_once_with(alert)


def test_notification_setting_accepts_yes(monkeypatch):
    monkeypatch.setenv("METEOR_NOTIFY", " yes ")
    assert notifications_enabled() is True


def test_notification_setting_defaults_to_disabled(monkeypatch):
    monkeypatch.delenv("METEOR_NOTIFY", raising=False)
    assert notifications_enabled() is False
```

Add the specified failure and patch-location tests around this core rather than turning every collaborator into a mock.

Key points

- Hidden clocks, randomness, environment state, and services make tests hard to reproduce.
- An explicit callable or small collaborator often creates the clearest seam.
- Choose a stub, fake, spy, or mock by the evidence required, not by fashion.
- `monkeypatch` changes process state temporarily and restores the real prior value.
- Patch the namespace where the system under test looks up the name.
- `Mock(spec=...)` catches some imaginary attributes; it does not prove a good test design.
- Assert state or returned output unless an interaction is itself a public obligation.
- Avoid real services, sleeps, broad builtin patches, and deep mock chains in focused tests.

- Use real deterministic boundaries such as `tmp_path` when they are simpler and more faithful than a double.

Continue learning

- pytest documentation: safely patch attributes, mappings, environment, and directories
- Python documentation: `unittest.mock`
- Python documentation: where to patch
- Next: Test at the Smallest Boundary That Can Fail

Test at the Smallest Boundary That Can Fail

Online lesson and runnable notebook

Match product risks to focused function, integration, process, and end-to-end boundaries; interpret statement and branch coverage; and remove flaky dependencies without chasing a universal test pyramid.

- **Level:** Python Foundations
- **Estimated time:** 4–5 hours
- **You will learn:** Select a test boundary from a concrete risk, combine fast diagnostic checks with a few broader proofs, and use coverage and flakiness evidence to improve—not grade—the suite.
- **Practice in:** A local installed project and terminal for direct, filesystem, coverage, and child-process checks

A pure-function test can prove Meteor Watch classifies 70 kph as red. It cannot prove the installed command starts from another directory. A subprocess smoke test can prove the command starts, reads input, writes output, and returns the expected status. When that broad test fails, it may not immediately reveal whether parsing, configuration, rendering, packaging, or process wiring is at fault.

No single level provides every kind of confidence. The useful question is not “How many unit tests should this pyramid contain?” It is “What risk could harm the user, and what is the smallest boundary that can actually expose it?”

Use these questions as you assemble a test portfolio:

- Which components must be real for the risk to exist?
- What is the smallest boundary whose failure would reveal that risk?
- Which faster check would make a broad failure easier to diagnose?
- What does an uncovered line or branch invite you to investigate?
- Which hidden dependency could make a result change without code changes?

1. Name boundaries by what they cross

Testing vocabulary varies between teams. Classify a test by its actual boundary rather than trusting a filename or label.

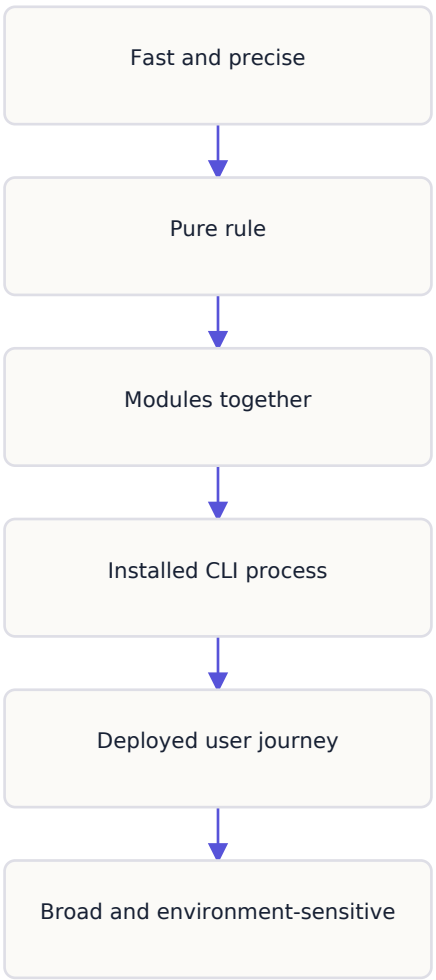
Boundary	Real parts	Replaced or controlled parts	Useful risk
focused function/component	one cohesive unit of behavior	external boundaries supplied as values or doubles	decision logic, validation, formatting
integration	two or more real local parts	destructive/remote edges may remain controlled	parser-to-model, model-to-file, configuration-to-core

Boundary	Real parts	Replaced or controlled parts	Useful risk
process/smoke	installed entry point and operating-system process	remote service normally controlled	imports, entry point, argv, streams, exit status
end-to-end/system	complete deployed user route	little or nothing inside the route	a critical journey across realistic infrastructure

“Unit” does not necessarily mean one Python function. A small class and its value object may form one cohesive component. “Integration” does not automatically mean a database or internet connection. Writing through a real `Path` into `tmp_path` integrates application code with the local filesystem.

A **smoke test** asks whether a broad route basically works, not whether every branch is correct. An **end-to-end** test follows a realistic user journey through the whole system. This foundations project has no deployed web service, so a child-process CLI test is its broadest meaningful boundary. Do not invent a browser merely to fill a taxonomy box.

Each wider boundary includes more real parts and can expose different risks, but failures generally cost more and have more possible causes.



2. Begin with a risk, not a target percentage

Create a small risk register:

Risk	Consequence	Smallest revealing boundary	Why narrower is insufficient
wind 70 classified amber	severe alert missed	pure `alert_level` call	function is the complete rule owner
UTF-8 station damaged in report	user cannot identify station	report writer + real temporary file	returned in-memory text does not exercise encoding/write path
configuration value never reaches classifier	wrong threshold used	config parser + application coordinator	parser alone and classifier alone do not prove wiring
console entry point missing after installation	command cannot start	installed subprocess	direct `main()` bypasses package metadata and process lookup
stderr mixed into JSON stdout	pipeline data corrupted	direct `main()` with separate streams, plus one process check	pure rendering does not prove adapter channel wiring

The smallest boundary is not always the fewest lines. It includes exactly the parts necessary for the failure mode. If the risk is console-script metadata, no amount of direct function testing can expose it.

3. Build a focused function layer

The pure rule remains fast and diagnostic:

PYTHON

```
import pytest

from meteor_watch.alerts import alert_level


@pytest.mark.parametrize(
    ("wind_kph", "visibility_km", "expected"),
    [
        pytest.param(39.9, 10, "green", id="below-amber"),
        pytest.param(40, 10, "amber", id="wind-at-amber"),
        pytest.param(69.9, 10, "amber", id="below-red"),
        pytest.param(70, 10, "red", id="wind-at-red"),
        pytest.param(45, 0.9, "red", id="red-outranks-amber"),
    ],
)
def test_alert_level_rules(wind_kph, visibility_km, expected):
    assert alert_level(wind_kph, visibility_km) == expected
```

If `wind-at-red` fails, the node identifies one comparison. This layer can cover many rules cheaply and without filesystem or process noise.

Focused does not mean isolated from every standard-library value. A pure function can accept a `datetime`, `Path`, dataclass, or dictionary. Replace a dependency only when crossing it would obscure or destabilize the behavior being investigated.

Checkpoint: place the test at a revealing boundary

Match risks to the smallest revealing boundary

Choose the test whose real parts can expose the named failure.

1. Which test can reveal that an installed console entry point is missing?

- **A.** A direct alert_level call
- **B.** Launch the installed command as a child process
- **C.** A mocked Path.write_text
- **D.** A dictionary equality assertion

2. Which boundary best checks that "serra-ç" survives a UTF-8 report write?

- **A.** A mock call-count assertion
- **B.** Only the risk classifier
- **C.** The real report writer with a tmp_path file
- **D.** A coverage percentage

3. What determines whether a test is an integration test?

- **A.** Its function begins with test_integration
- **B.** It has at least ten assertions
- **C.** It always contacts the internet
- **D.** The real component boundaries it actually connects

Write your choices before checking the answer key at the end of this book.

4. Integrate real local collaborators where the risk crosses them

Suppose `load_thresholds` reads JSON and returns validated settings:

PYTHON
```
import json


def load_thresholds(path):
    data = json.loads(path.read_text(encoding="utf-8"))
    amber = float(data["amber_wind_kph"])
    red = float(data["red_wind_kph"])
    if not 0 <= amber < red:
        raise ValueError("thresholds must satisfy 0 <= amber < red")
    return {"amber": amber, "red": red}
```

An integration test uses a real temporary JSON file:

PYTHON
```
def test_load_thresholds_reads_and_validates_json(tmp_path):
    path = tmp_path / "meteor.json"
    path.write_text(
        '{"amber_wind_kph": 40, "red_wind_kph": 70}',
        encoding="utf-8",
```

PYTHON (CONTINUED)

```
    )

    assert load_thresholds(path) == {"amber": 40.0, "red": 70.0}
```

This crosses `Path`, text decoding, JSON parsing, key lookup, numeric conversion, and validation. Separate focused tests can diagnose invalid ordering or missing keys. The integration test proves those pieces connect for one ordinary file.

Do not replace every `Path` method with a mock simply to call the test a unit test. The real boundary is fast, deterministic, and central to the risk. Use a fake or patch when the collaborator is remote, destructive, unavailable, or otherwise unsuitable for a normal suite.

Integrate the coordinator without starting a process

The Unit 12 pattern passes streams and arguments explicitly:

PYTHON

```
from io import StringIO


def main(argv, stdout, stderr):
    if argv != ["classify"]:
        stderr.write("usage: meteor-watch classify\n")
        return 2
    stdout.write("ridge-7|red\n")
    return 0


def test_main_keeps_data_and_diagnostics_separate():
    stdout = StringIO()
    stderr = StringIO()

    status = main(["classify"], stdout, stderr)

    assert status == 0
    assert stdout.getvalue() == "ridge-7|red\n"
    assert stderr.getvalue() == ""
```

This connects parser/coordinator/rendering responsibilities in one process but does not prove an installed command, `sys.argv`, or process status translation. It is faster and easier to diagnose than a child process, so keep it even when one smoke test is added.

5. Cross the operating-system boundary once for the route that matters

After installing the package, run the console command with `subprocess.run`:

PYTHON

```
import os
import subprocess


def test_installed_command_classifies_from_another_directory(tmp_path):
    environment = os.environ.copy()
    result = subprocess.run(
        ["meteor-watch", "classify"],
        cwd=tmp_path,
        env=environment,
        text=True,
```

PYTHON (CONTINUED)

```
        capture_output=True,
        check=False,
    )

    assert result.returncode == 0
    assert result.stdout == "ridge-7|red\n"
    assert result.stderr == ""
```

This test proves:

- the installed executable is discoverable in the test environment;
- it does not depend on the repository as current directory;
- process argument routing reaches the command;
- stdout and stderr remain separate; and
- `main()` becomes process status 0.

It does not prove every option or validation branch. Keep one or a few process routes chosen by packaging and channel risk. Test detailed command grammar by calling parser or `main()` directly.

If the project supports `python -m meteor_watch`, test that route separately only if it is promised to users. Do not duplicate every assertion across console-script and module entry routes without a reason.

Make the environment explicit

The subprocess inherits environment variables unless you pass a controlled mapping. Copy `os.environ`, remove or replace application-specific keys, and avoid logging secrets in a failure. Running from `tmp_path` exposes accidental current-directory imports and resource paths. :::

6. Treat end-to-end checks as selected journeys

If Meteor Watch later becomes a deployed service, an end-to-end test might submit an observation through its public interface and verify the alert seen by a user. That route is valuable for integration risk but may involve deployment, credentials, queues, networks, and slower diagnosis.

Choose a few critical journeys:

- the simplest successful user goal;
- one high-consequence rejection;
- one boundary whose deployment wiring has failed before; and
- perhaps a health or smoke route after deployment.

Do not move every rule assertion to the broadest layer. A hundred browser or remote tests for numeric thresholds would be slow, fragile, and difficult to diagnose compared with a parametrized pure-function table.

Checkpoint: assemble complementary evidence

Combine focused, integration, and process evidence

Choose the boundary that adds a distinct kind of confidence.

1. What does calling main(argv, StringIO(), StringIO()) check that a pure classifier test does not?

- **A.** Coordinator routing, rendering, channels, and returned status in one process
- **B.** Installed console-script metadata
- **C.** The complete deployed environment
- **D.** Every possible argument

2. One subprocess smoke test passes. Which claim is too strong?

- **A.** The selected command route started
- **B.** Every command option and data rule is correct
- **C.** The observed stdout and status matched this case
- **D.** The command worked from the selected temporary directory

3. Why keep fast focused tests when a broad process test already exists?

- **A.** Pytest requires duplicate assertions
- **B.** Process tests cannot fail
- **C.** Focused failures locate rule defects more precisely and can cover many cases cheaply
- **D.** Focused tests prove installation

Write your choices before checking the answer key at the end of this book.

7. Use statement coverage to find unvisited code

The repository already includes pytest-cov. Run:

```bash
python -m pytest --cov=meteor_watch --cov-report=term-missing
```

A report might show:

```text
Name                            Stmts   Miss  Cover   Missing
---------------------------------------------------------------
src/meteor_watch/alerts.py         12      2    83%   18-19
src/meteor_watch/config.py         10      1    90%   14
---------------------------------------------------------------
TOTAL                              22      3    86%
```

Statement coverage answers: “Which executable lines did this run visit?” Open lines 18–19 and ask:

- Is this behavior part of the supported contract?
- Which risk would occur if it were wrong?
- Is the line unreachable and removable?
- Is the code defensive for an impossible state that should instead be prevented by design?
- What focused test boundary can execute it meaningfully?

Do not write a content-free test merely to turn the line green. If the missing line logs an unsupported internal state, create the state only if the public contract permits it or redesign the state.

8. Add branch evidence when one line has more than one destination

Statement coverage can visit every line without taking every decision outcome:

```python
def report_label(risk, compact=False):
    label = risk.upper() if compact else f"{risk.upper()} risk"
    return label
```

One test with `compact=False` executes both source lines, producing 100% statement coverage. It does not check the true destination of the conditional expression.

Run branch coverage:

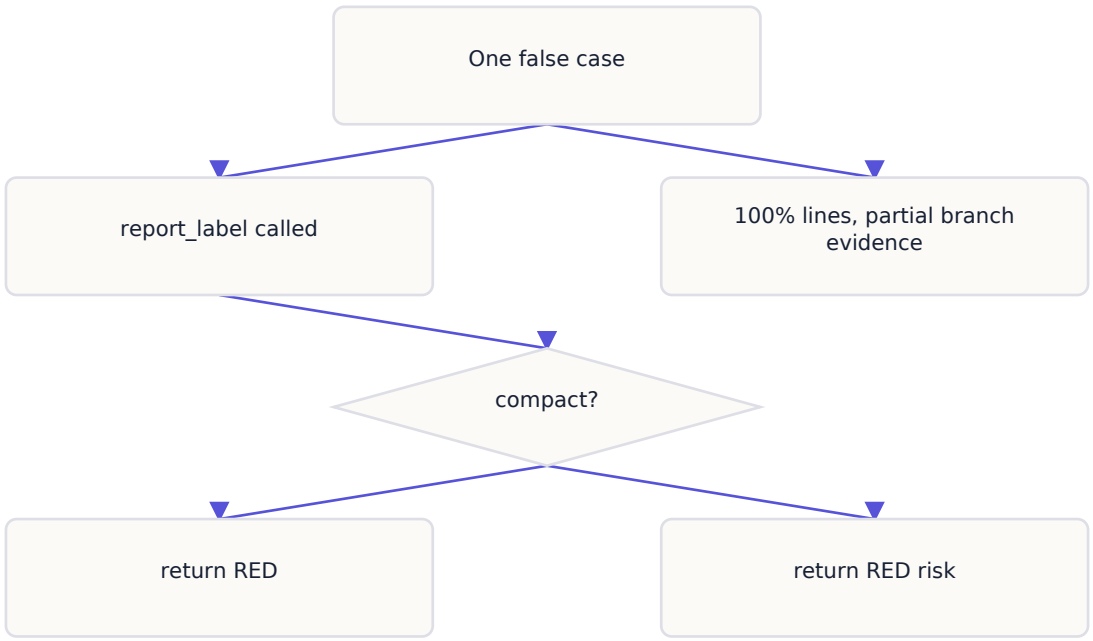
```bash
python -m pytest \
  --cov=meteor_watch \
  --cov-branch \
  --cov-report=term-missing
```

Then add a test because compact output is a promised route:

```python
def test_report_label_compact_omits_suffix():
    assert report_label("red", compact=True) == "RED"
```

Branch coverage records possible transitions between lines and can identify a decision destination that did not occur. It still cannot tell whether the assertion is meaningful, whether boundaries are correct, or whether a missing requirement exists.

Executing the decision line is statement evidence. Taking both destinations is branch evidence. Neither decides whether the expected strings match the real contract.



compact? → return RED: true

compact? → return RED risk: false

9. Reject coverage theatre

Coverage is not correctness. This test visits the line without checking it:

```
PYTHON
def test_report_label_runs():
    report_label("red", compact=True)
```

The percentage can increase while protection does not. Likewise, a test that asserts `result is not None` may visit every classifier branch without checking the right labels.

Use coverage in review:

1. run meaningful behavior tests;
2. inspect missing statements and branches;
3. connect each gap to a contract or risk;
4. add a focused test, remove unreachable code, or record why the gap is intentionally outside this suite; and
5. review the new assertion by the failure it would catch.

A project may use a minimum coverage threshold to prevent a large accidental drop. That threshold is a guardrail, not a grade and not a reason to demand 100% from every module. Unit 14 will discuss automated quality gates; here the learning goal is interpretation.

10. Remove hidden causes of flaky tests

A **flaky test** sometimes passes and sometimes fails without an intentional code or contract change. Common causes include:

- shared mutable state or order dependence;
- current time and timing windows;
- unseeded or globally seeded randomness;
- real network or service availability;
- sleeps used for coordination;
- temporary ports, paths, or files assumed to have fixed names;
- process-global environment or working-directory changes without restoration;
- concurrency and event ordering; and
- tests that depend on performance timing rather than functional results.

Retries can collect evidence, but automatically rerunning until green can hide a defect. Reproduce the failure, record its seed/order/environment when available, and control the responsible dependency.

This test is flaky around midnight:

```python
from datetime import datetime


def test_alert_uses_today():
    alert = build_live_alert("ridge-7")
    assert alert["day"] == datetime.now().date().isoformat()
```

If the date changes between the production and test calls, it fails. Inject one clock value and derive both behavior and expected date from the contract:

```python
from datetime import datetime, timezone


def test_alert_uses_injected_day():
    moment = datetime(2035, 6, 1, 23, 59, tzinfo=timezone.utc)

    alert = build_alert_for_moment("ridge-7", moment)

    assert alert["day"] == "2035-06-01"
```

Do not add a one-second tolerance or repeat the assertion. Remove the race.

Checkpoint: interpret coverage and instability

Turn coverage and flaky failures into questions

Choose the conclusion supported by the observed evidence.

1. A module reports 100% statement coverage. Which conclusion is justified?

- **A.** Every requirement is correct
- **B.** Every branch and input is proven
- **C.** The program has no defects
- **D.** Every counted executable statement ran at least once in that measured run

2. Why add –cov-branch to the report_label example?

- **A.** To reveal whether both compact and non-compact decision destinations ran
- **B.** To make expected strings correct automatically
- **C.** To launch a subprocess
- **D.** To eliminate the need for assertions

3. A date test fails only around midnight because production and test read the clock separately. What is the strongest repair?

- **A.** Retry until it passes
- **B.** Inject one controlled moment used by the behavior
- **C.** Increase coverage to 100%
- **D.** Skip the test at night

Write your choices before checking the answer key at the end of this book.

11. Design the smallest useful confidence portfolio

Create a test portfolio for these Meteor Watch risks:

1. wind and visibility boundaries may be classified incorrectly;
2. non-ASCII station names may be damaged in a report file;
3. invalid threshold ordering may reach the classifier;
4. stderr diagnostics may contaminate stdout data;
5. the installed command may depend on the project directory;
6. compact formatting has an untested branch; and
7. a date assertion sometimes fails around midnight.

For each risk, record:

Risk	Smallest revealing boundary	Real parts	Controlled parts	Expected failure evidence

Then implement:

- focused parametrized classification cases;
- one `tmp_path` UTF-8 integration;
- one configuration validation integration;
- one direct `main()` streams/status test;
- one installed subprocess smoke test launched from `tmp_path`;
- one compact branch assertion; and
- one deterministic injected-time test.

Run statement and branch coverage. Add a test only when an uncovered location maps to a supported behavior or recorded risk. Identify one redundant broad test that can be removed because a smaller test protects the same risk more precisely, but do not remove the only installation proof.

Hint 1: no boundary is universally best

The classifier belongs at the function boundary. Encoding needs a real file. Entry-point installation needs a process. Match the real parts to the failure mode rather than forcing every risk into one level.

Hint 2: use two checks around the command boundary

Call `main(argv, stdout, stderr)` directly for several detailed cases. Keep one subprocess smoke check for installed entry point, separate channels, status, and working-directory independence.

Hint 3: ask what an uncovered branch means

Open the missing location. If compact formatting is public, add the exact compact assertion. If the branch is unreachable because validation prevents it, consider simplifying the code rather than manufacturing an impossible test.

Show one justified portfolio

One defensible mapping is:

- boundary rules → parametrized direct `alert_level` tests;

- UTF-8 persistence → real writer plus `tmp_path` readback;
- threshold ordering → real JSON loader and validator with a temporary file;
- stream separation → direct `main()` with two `StringIO` objects;
- installation/current directory → one `subprocess.run` smoke test from `tmp_path`;
- compact route → one direct exact-result test prompted by branch evidence; and
- midnight race → injected fixed datetime with no tolerance or retry.

The final explanation should say why each broader boundary exists and which detailed cases stay at the faster layer.

Key points

- Classify a test by the real boundary it crosses, not by its filename.
- Begin with product risk and choose the smallest boundary capable of exposing it.
- Focused tests provide cheap depth and precise diagnosis.
- Real local integrations such as `tmp_path` can be simpler than mocks.
- Direct `main()` tests cover detailed adapter behavior; a small number of subprocess tests protect installation and process wiring.
- End-to-end checks should represent selected critical journeys, not every branch.
- Statement coverage reveals visited lines; branch coverage reveals decision destinations.
- Coverage raises questions and can detect drops, but it does not prove correctness or replace assertions.
- Flaky tests reveal uncontrolled state, time, randomness, services, order, or coordination; remove the cause instead of retrying until green.

Continue learning

- pytest documentation: good integration practices
- pytest-cov documentation: reporting coverage from pytest
- Coverage.py documentation: branch coverage
- pytest documentation: flaky tests
- Next: Preserve Bugs and Explore General Rules

Preserve Bugs and Explore General Rules

Online lesson and runnable notebook

Turn minimal failures into named regressions, complete an honest red-green-refactor cycle, and use Hypothesis strategies and shrinking to test invariants beyond a few remembered examples.

- **Level:** Python Foundations
- **Estimated time:** 4–5 hours
- **You will learn:** Preserve a discovered failure, work through red–green–refactor without changing the contract mid-cycle, and use generated examples and shrinking to investigate a general rule.
- **Practice in:** A local pytest project with Hypothesis installed; notebook users can run direct examples after installing the same dependency

A hand-picked suite can be thoughtful and still miss a strange input. A user may type a tab where you expected a space, combine Unicode letters, repeat a separator, or provide a number near a boundary you did not remember. Once such a failure is found, a named regression prevents its quiet return. A property test can then explore the broader rule that the one example violated.

This lesson connects three practices:

- **regression testing:** keep evidence for a defect that once occurred;
- **test-driven development (TDD):** add one promised behavior through red, green, and refactor; and
- **property-based testing:** generate many examples from a described input space and check an invariant for all examples explored in the run.

They are complementary. TDD does not discover an unclear contract for you. Generated examples do not replace readable ordinary examples. A regression test does not prove all related inputs. Use each for the question it answers.

As you work, answer:

- How can a large failure be reduced without removing the behavior that causes it?
- What evidence distinguishes a useful red test from a broken test setup?
- Which invariants express more than a list of expected examples?
- How do strategies define valid data and shrinking find a smaller failure?
- When should a generated counterexample become a descriptive regression test?

1. Reduce a failure before preserving it

Meteor Watch normalizes a station callsign for matching:

PYTHON

```
def normalize_callsign(text):
    """Trim outside whitespace, collapse inside whitespace, and uppercase."""
    return " ".join(text.strip().split(" ")).upper()
```

It works for one ordinary string:

```python
assert normalize_callsign(" ridge seven ") == "RIDGE SEVEN"
```

But a user report contains a tab:

```python
assert normalize_callsign("ridge\tseven") == "RIDGE SEVEN"
```

The result is `"RIDGE\tSEVEN"` because `split(" ")` recognizes only the literal space separator. The original report may have contained several records and a large traceback. Reduce it while preserving the failure:

1. remove unrelated records;
2. remove unrelated fields;
3. shorten the station text;
4. replace multiple whitespace forms one at a time; and
5. stop when removing the tab makes the failure disappear.

`"a\tb"` is a smaller counterexample to the same rule:

```python
def test_tab_between_callsign_words_is_collapsed_regression():
    assert normalize_callsign("a\tb") == "A B"
```

Run this test against the buggy implementation. It must fail because a tab remains, not because the package import or fixture setup is broken. Then repair the implementation with `text.split()`, whose no-argument form splits on runs of Unicode whitespace:

```python
def normalize_callsign(text):
    """Trim outside whitespace, collapse inside whitespace, and uppercase."""
    return " ".join(text.split()).upper()
```

Keep the regression after the repair. Its descriptive name records why a tab matters even if a future refactor changes the implementation again.

Record enough history, not an incident report in the test

The name and a short comment may be enough:

```python
def test_tab_between_callsign_words_is_collapsed_regression():
    # Tabs appeared in station exports from older field terminals.
    assert normalize_callsign("a\tb") == "A B"
```

Issue links can live in a comment when the history is useful and stable. Do not paste a long ticket into the test. The executable input, expected result, and reason should remain understandable if the link disappears.

2. Make red fail for the intended missing behavior

TDD is often summarized as:

1. **red:** write a small test for one missing or changed behavior;
2. **green:** write the smallest clear production change that satisfies it; and
3. **refactor:** improve structure while all tests remain green.

The color is not enough. A test that is red because `meteor_watch` cannot import does not demonstrate missing behavior. Read the report and confirm the expected assertion or exception is responsible.

Meteor Watch needs a new `"critical"` result when wind is at least 110 kph. The current contract has only green, amber, and red. Start with the changed requirement table:

Wind	Visibility	Expected
109.9	10	red
110	10	critical

Write the boundary test first:

```python
def test_wind_one_hundred_ten_begins_critical_alert():
    assert alert_level(110, 10) == "critical"
```

Run only that node. The useful red evidence is `assert 'red' == 'critical'`. If it passes before production changes, either behavior already exists or the test is not reaching the intended code.

Checkpoint: preserve a meaningful failure

Reduce failures and begin with useful red evidence

Choose the regression or TDD step that preserves the actual behavior.

1. A long field report fails because a callsign contains one tab. Which regression input is most useful?

- **A.** The entire production archive
- **B.** An empty string that passes
- **C.** A short string such as "a\tb" that still fails
- **D.** A random unrelated filename

2. The new critical-boundary test is red because meteor_watch cannot be imported. What should happen next?

- **A.** Implement critical behavior immediately
- **B.** Change expected to red
- **C.** Mark the test xfail forever
- **D.** Repair the environment/import so red can expose the intended missing behavior

3. What should happen to the tab regression after normalize_callsign is fixed?

- **A.** Keep it as lasting protection for the past failure
- **B.** Delete it because the defect is gone
- **C.** Replace it with assert True
- **D.** Move the tab into production code

Write your choices before checking the answer key at the end of this book.

3. Write the smallest clear green change

Add the critical branch before red:

```python
def alert_level(wind_kph, visibility_km):
    if wind_kph < 0:
        raise ValueError("wind must be non-negative")
    if visibility_km < 0:
        raise ValueError("visibility must be non-negative")
    if wind_kph >= 110:
        return "critical"
    if wind_kph >= 70 or visibility_km < 1:
        return "red"
    if wind_kph >= 40 or visibility_km < 5:
        return "amber"
    return "green"
```

Run the new node, then the existing boundary table. The new test may be green while an old invariant test still allows only `{"green", "amber", "red"}`. That old failure is valuable: the public vocabulary changed and its test must be updated deliberately.

Do not implement speculative categories, configuration, or notification rules that no current contract asks for. “Smallest” does not mean cryptic; it means no unrequested design expansion.

4. Refactor only while the behavior stays green

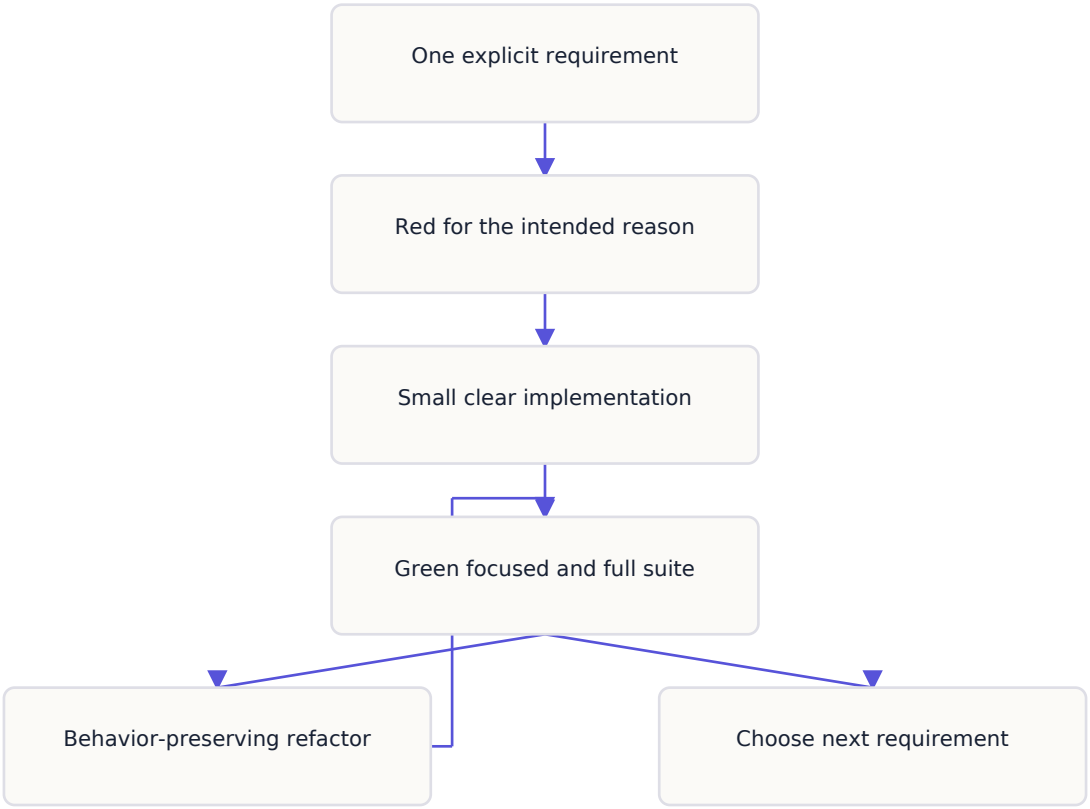
The branch ordering is readable, but duplicated validation can move into a helper if several functions use it:

```python
def validate_measurements(wind_kph, visibility_km):
    if wind_kph < 0:
        raise ValueError("wind must be non-negative")
    if visibility_km < 0:
        raise ValueError("visibility must be non-negative")


def alert_level(wind_kph, visibility_km):
    validate_measurements(wind_kph, visibility_km)
    if wind_kph >= 110:
        return "critical"
    if wind_kph >= 70 or visibility_km < 1:
        return "red"
    if wind_kph >= 40 or visibility_km < 5:
        return "amber"
    return "green"
```

Run the complete suite after the extraction. Refactoring means changing structure without intentionally changing public behavior. If you add another category during this step, you have mixed a feature with the refactor and made failures harder to interpret.

Each TDD phase has different evidence. Refactor returns to green after a structural change; it does not quietly add a second requirement.



Behavior-preserving refactor → Green focused and full suite: stay green

When TDD helps—and when to explore first

TDD is useful when a small behavior can be stated before implementation:

- a parser accepts a documented record;
- a boundary changes at a known value;
- a defect has a reproducible input;
- a new function has a clear contract; or
- an adapter must translate a known status.

Exploration may come first when the problem, library, or user need is unclear. Use a notebook or disposable spike to learn. Then discard or clean the spike, write the contract learned from it, and add durable tests. Pretending an uncertain experiment was known in advance turns TDD into ceremony.

5. State properties that cover more than remembered examples

Examples say what should happen for specific inputs. A **property** says what relationship should hold across a range of inputs.

Useful property shapes include:

- **idempotence:** normalizing twice equals normalizing once;
- **round trip:** encoding then decoding returns the original supported value;
- **bounds:** a risk score always remains within its promised interval;
- **monotonicity:** increasing wind while holding visibility fixed cannot reduce severity;
- **invariance:** adding outside whitespace does not change a normalized callsign; and
- **model agreement:** a production result equals a smaller independently trusted model.

For callsigns:

```
PYTHON
def test_normalize_callsign_is_idempotent_for_one_example():
    once = normalize_callsign("  Ridge\tSeven ")
    twice = normalize_callsign(once)

    assert twice == once
```

That is a property-shaped example. It communicates the invariant but explores only one string. Hypothesis can generate many strings from a strategy.

6. Install Hypothesis and generate valid text

Install Hypothesis in the practice environment:

```
BASH
python -m pip install hypothesis
python -c "import hypothesis; print(hypothesis.__version__)"
```

The repository declares Hypothesis as a development dependency, so a Poetry development installation includes it.

Import `given` and strategies, conventionally named `st`:

```
PYTHON
from hypothesis import given
from hypothesis import strategies as st


@given(st.text())
def test_normalize_callsign_is_idempotent(text):
    once = normalize_callsign(text)
    assert normalize_callsign(once) == once
```

For each generated `text`, Hypothesis calls the test and checks the assertion. The exact number of attempted examples is configurable and may include database replays or targeted cases; do not assert that a run always uses one fixed list.

The empty string is valid under `st.text()`. Decide whether it belongs to the normalizer contract. If empty callsigns should be rejected, test a validator with a strategy that represents its accepted domain rather than hiding all empty failures.

Constrain the strategy to the contract

If a station callsign accepts letters, digits, spaces, tabs, and hyphens with a maximum raw length of 40:

```python
from hypothesis import strategies as st


callsign_text = st.text(
    alphabet=st.characters(
        categories=("L", "N"),
        include_characters=" \t-",
    ),
    min_size=1,
    max_size=40,
)
```

Use it:

```python
from hypothesis import given


@given(callsign_text)
def test_normalization_is_idempotent_for_callsign_text(text):
    normalized = normalize_callsign(text)
    assert normalize_callsign(normalized) == normalized
```

Constraining generation directly is usually clearer and faster than generating arbitrary huge strings and rejecting most with `.filter(...)` or `assume(...)`. Use `assume` when a relationship between generated values is genuinely hard to encode, and watch for health-check evidence that too many examples are discarded.

Checkpoint: choose properties and strategies

Express general rules with generated inputs

Choose the property or strategy that matches the supported input domain.

1. Which assertion expresses idempotent callsign normalization?

- **A.** normalize(text) != text
- **B.** normalize(normalize(text)) == normalize(text)
- **C.** len(normalize(text)) == len(text) for all text
- **D.** normalize(text) is text

2. The contract accepts at most 40 letters, digits, spaces, tabs, or hyphens. What is the clearest generation approach?

- **A.** Generate arbitrary Python objects and ignore errors
- **B.** Generate million-character strings then filter them
- **C.** Build a text strategy with the accepted alphabet and size bounds
- **D.** Use one hand-picked string only

3. What should happen to readable boundary and regression examples after adding a property test?

- **A.** Delete every example
- **B.** Mark them all skipped

- **C.** Turn them into production comments
- **D.** Keep examples that communicate specific rules and history alongside the broader property

Write your choices before checking the answer key at the end of this book.

7. Read the counterexample and the shrink

Restore the buggy `split(" ")` normalizer and run the idempotence property. Idempotence may still pass because preserving a tab twice is stable. This is an important lesson: a true but weak property can miss the defect.

Add an invariant from the contract: normalized output contains no whitespace other than single spaces between non-space tokens.

PYTHON
```
from hypothesis import given


@given(callsign_text)
def test_normalized_callsign_has_only_single_spaces(text):
    normalized = normalize_callsign(text)
    assert "\t" not in normalized
    assert "  " not in normalized
    assert normalized == normalized.strip()
```

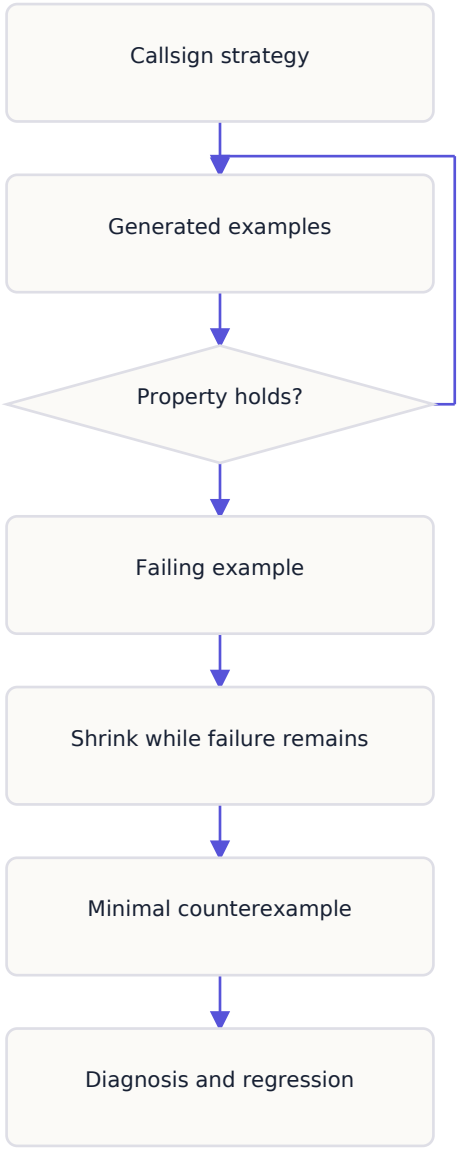
Hypothesis may report a falsifying example such as `text='0\t0'`. It tries to **shrink** a failure: find a simpler value that still makes the assertion false. A small counterexample helps reveal the rule and reduces debugging noise.

Representative output:

TEXT
```
E   AssertionError: assert '\t' not in '0\t0'
E     '\t' is contained here:
E       0   0
E   Falsifying example: test_normalized_callsign_has_only_single_spaces(
E       text='0\t0',
E   )
```

Do not assume the first shown value is the only possible failure. Read the assertion, strategy domain, and minimized example. Reproduce it as a direct call, confirm `split(" ")` is responsible, and repair with `split()`.

Generation searches the described domain. Shrinking keeps the failure while removing irrelevant complexity, producing a more useful debugging input.



Property holds? → Generated examples: yes

Property holds? → Failing example: no

Preserve important generated failures

Hypothesis maintains an examples database that normally replays failures. Still add a named ordinary regression when the value represents a meaningful bug or communicates a boundary:

```python
def test_tab_between_callsign_words_is_collapsed_regression():
    assert normalize_callsign("0\t0") == "0 0"
```

The property continues exploring variations. The named regression explains the specific obligation without depending on database state.

8. Build numeric strategies around real boundaries

Properties can complement exact alert examples. Define severity ordering:

```python
SEVERITY = {
    "green": 0,
    "amber": 1,
    "red": 2,
    "critical": 3,
}
```

Increasing wind should not reduce severity while visibility is fixed:

```python
from hypothesis import given
from hypothesis import strategies as st


@given(
    lower=st.floats(
        min_value=0,
        max_value=200,
        allow_nan=False,
        allow_infinity=False,
    ),
    increase=st.floats(
        min_value=0,
        max_value=200,
        allow_nan=False,
        allow_infinity=False,
    ),
    visibility=st.floats(
        min_value=0,
        max_value=20,
        allow_nan=False,
        allow_infinity=False,
    ),
)
def test_more_wind_never_reduces_alert_severity(
    lower,
    increase,
    visibility,
):
    higher = lower + increase
    lower_severity = SEVERITY[alert_level(lower, visibility)]
    higher_severity = SEVERITY[alert_level(higher, visibility)]

    assert higher_severity >= lower_severity
```

The strategy excludes NaN and infinity because the current numeric contract accepts finite non-negative measurements. If production should reject non-finite floats, add explicit examples and a separate property for that validation contract.

Properties can be wrong. If increasing wind legitimately changes another mode, the monotonic claim may overconstrain behavior. Review the property like any expected value.

9. Reproduce generated failures without hard-coding a random seed

Hypothesis reports a counterexample and saves it in its database. Rerun the focused node first. During investigation:

1. copy the minimal value into a direct function call;
2. confirm the property fails for the documented reason;

3 simplify production or the property one change at a time;

4 retain a named regression if the example communicates a real defect; and

5 rerun the property and full suite.

Hypothesis offers settings, seeds, and explicit `@example` values, but do not pin a seed merely to turn property-based exploration into one fixed random list. Use `@example(...)` when a known case should always accompany generated cases; use a normal regression test when a descriptive node is more helpful.

Checkpoint: interpret shrinking and combine evidence

Use a minimal counterexample without overclaiming

Choose the next action supported by generated failure evidence.

1. What is Hypothesis shrinking trying to produce after a failure?

- **A.** A simpler input that still causes the same property failure
- **B.** The largest possible benchmark
- **C.** A passing replacement expected value
- **D.** A coverage threshold

2. The buggy normalizer preserves tabs, yet the idempotence property passes. What does that show?

- **A.** Hypothesis cannot generate tabs
- **B.** A property can be true but too weak to express the missing contract rule
- **C.** All generated tests are flaky
- **D.** The implementation is correct

3. After a minimal tab counterexample exposes a real bug, what is the strongest lasting combination?

- **A.** Delete the property after fixing it
- **B.** Retry with random seeds only
- **C.** Keep the property and add a descriptive tab regression
- **D.** Ignore the example because it is small

Write your choices before checking the answer key at the end of this book.

10. Shrink the strange callsign

Build a final lab around `normalize_callsign` and `alert_level`:

1 begin with the buggy `split(" ")` normalizer;

2 keep ordinary examples for trimming, casing, repeated spaces, and hyphens;

3 add an idempotence property and explain why it does not expose every whitespace bug;

4 define a bounded callsign strategy including letters, digits, spaces, tabs, and hyphens;

5 add a property that normalized output has no tabs, no doubled spaces, and no outside whitespace;

6 retain Hypothesis’s minimal counterexample and reduce it as a direct call;

7 add a named tab regression, verify it is red, and fix production with `split()`;

8 add the critical wind requirement through a separate red–green–refactor cycle;

9 add the monotonic-severity property for finite non-negative measurements; and

10 run ordinary examples, regressions, properties, and the complete project from a clean process.

Your evidence log should distinguish the regression failure, TDD boundary failure, and property counterexample. Do not describe one as proof of the others.

Hint 1: make the whitespace property stricter than idempotence

Assert that `"\t"` and `"  "` are absent and that `normalized == normalized.strip()`. The buggy result can be stable under a second call while still containing forbidden whitespace.

Hint 2: constrain generation at the strategy

Use `st.text` with an alphabet containing letters, digits, `" "`, `"\t"`, and `"-"`, plus `min_size=1` and `max_size=40`. Avoid generating arbitrary objects and discarding nearly all of them.

Hint 3: keep the two red phases separate

Finish and verify the normalization regression before starting the new critical-alert requirement. A refactor happens with the existing suite green; the next feature begins with a new intentional red node.

Show a complete normalization property group

PYTHON

```
from hypothesis import given
from hypothesis import strategies as st

from meteor_watch.callsigns import normalize_callsign


callsign_text = st.text(
    alphabet=st.characters(
        categories=("L", "N"),
        include_characters=" \t-",
    ),
    min_size=1,
    max_size=40,
)


def test_callsign_trims_collapses_and_uppercases():
    assert normalize_callsign("  ridge   seven ") == "RIDGE SEVEN"


def test_tab_between_callsign_words_is_collapsed_regression():
    assert normalize_callsign("0\t0") == "0 0"


@given(callsign_text)
def test_normalization_is_idempotent(text):
    once = normalize_callsign(text)
    assert normalize_callsign(once) == once


@given(callsign_text)
def test_normalized_callsign_uses_single_spaces(text):
```

PYTHON (CONTINUED)

```
    normalized = normalize_callsign(text)
    assert "\t" not in normalized
    assert "  " not in normalized
    assert normalized == normalized.strip()
```

The repaired implementation is intentionally small:

PYTHON

```
def normalize_callsign(text):
    """Trim, collapse whitespace, and uppercase a station callsign."""
    return " ".join(text.split()).upper()
```

Add the separate critical threshold and monotonic-severity work from the lab contract before declaring the lesson complete.

Key points

- Reduce a real failure to the smallest input that preserves its cause.
- Keep meaningful failures as descriptive regression tests after the repair.
- TDD red must fail for the intended missing behavior, not a broken environment.
- Green is the smallest clear implementation of the current requirement; refactor changes structure while the complete suite remains green.
- Explore first when the problem is unclear, then turn what you learned into a contract and durable tests.
- Properties express relationships such as idempotence, round trips, bounds, and monotonicity across generated examples.
- Strategies should describe the supported input domain directly.
- Shrinking searches for a simpler value that still fails.
- A passing weak property can miss a defect; review the property as carefully as an expected example.
- Keep communicative examples, named regressions, and broader properties together when each supplies distinct evidence.

Continue learning

- Hypothesis documentation: quick start with pytest, strategies, and shrinking
- Hypothesis documentation: details of generated examples and failures
- pytest documentation: parametrization and generated cases
- Next: Clear the Buggy Spaceport for Launch

Unit Challenge: Clear the Buggy Spaceport for Launch

Online lesson and runnable notebook

Build a layered pytest suite for a playful spaceport dispatcher, expose six contract-backed defects, repair them one at a time, and leave boundary, file, process, regression, and property evidence.

- **Level:** Python Foundations · Unit 13 challenge
- **Estimated time:** 60–120 minutes
- **Outcome assessed:** Turn a public contract into an isolated, readable pytest portfolio that exposes real defects at function, file, collaborator, and process boundaries.
- **Evidence:** Approximately 30 collected cases, six red-to-green repairs, branch-coverage questions, one minimal generated counterexample, a clean process smoke test, and one debugging record

1. Receive the launch inspector’s warning

Tiny research ships are queuing at **Aurora Pocket Spaceport**. The dispatcher normalizes callsigns, calculates landing fees, stamps clearances, stores a JSON manifest, and exposes one small command. It looks ready—until the launch inspector reports six strange observations:

- tabs sometimes survive inside callsigns;
- a cargo pod exactly on a fee boundary receives the lower price;
- `True` can be accepted as a mass of one kilogram;
- clearance timestamps ignore the spaceport’s controlled clock;
- a saved manifest can quietly lose all but its first clearance; and
- an invalid command writes its diagnostic into the data channel.

Your job is to build the protection system before opening the launch gates. Every defect is tied to the public contract below. There are no secret trivia rules. Write a red test for one observation, confirm its failure phase and values, repair one production cause, and rerun the focused node before moving on.

Act as an inspector, not a guesser

Do not rewrite the module from scratch or change expected results to match its current output. Let each public rule produce a focused failure. Keep the smallest useful report, one repair, and the clean rerun as evidence. :::

The finished result should feel satisfying: one compact suite stops six shape-shifting bugs from boarding the ships again.

2. Translate the spaceport rules into observable behavior

Callsigns

`normalize_callsign(text)` must:

1 accept a string;

2. trim outside whitespace;
3. collapse every inside run of Unicode whitespace—including spaces and tabs—to one ordinary space;
4. uppercase letters;
5. preserve hyphens and digits; and
6. raise `ValueError("callsign must not be empty")` when no non-whitespace text remains.

Examples:

Input	Result
`" nova 7 "`	`"NOVA 7"`
`"nova\t7"`	`"NOVA 7"`
`"lx-42"`	`"LX-42"`

Landing fees

`landing_fee(mass_kg, hazardous=False)` must reject booleans, non-numeric values, and negative values. For accepted finite numeric values:

Mass	Base fee
`0 <= mass < 100`	5 credits
`100 <= mass < 500`	12 credits
`mass >= 500`	25 credits

Hazardous cargo adds exactly 7 credits after the base tier is selected.

Clearances

`build_clearance(record, clock)` receives a mapping with `callsign`, `mass_kg`, and optional `hazardous`. It must return a new dictionary containing:

- normalized `callsign`;
- original numeric `mass_kg`;
- Boolean `hazardous`;
- calculated `fee`; and
- `created_at` from exactly one call to the supplied zero-argument `clock`, formatted with `datetime.isoformat()`.

It must not mutate `record`.

Manifests

`save_clearances(path, records)` writes **every** record as one UTF-8 JSON array and returns the number written. `load_clearances(path)` returns the complete list. An empty list and non-ASCII callsign must round-trip. Invalid JSON is allowed to raise `json.JSONDecodeError` with its original context.

Command boundary

The module supports:

```text
python -m spaceport_dispatch fee MASS [--hazardous]
```

On success it writes only the integer fee plus a newline to stdout, writes nothing to stderr, and returns status 0. Invalid mass writes `spaceport-dispatch: <message>` plus a newline to stderr, leaves stdout empty, and returns status 2.

3. Start from the contract

Create a disposable project:

```text
spaceport-test-lab/
├── spaceport_dispatch.py
└── tests/
    └── test_spaceport_dispatch.py
```

Install the tools in an isolated environment if they are not already present:

```bash
python -m pip install pytest pytest-cov hypothesis
```

Copy this **deliberately buggy but syntactically valid** module into `spaceport_dispatch.py`:

```python
import argparse
import json
import math
import sys
from datetime import datetime, timezone


def normalize_callsign(text):
    """Trim, collapse whitespace, and uppercase a non-empty callsign."""
    if not isinstance(text, str):
        raise TypeError("callsign must be text")
    normalized = " ".join(text.strip().split(" ")).upper()
    if not normalized:
        raise ValueError("callsign must not be empty")
    return normalized


def landing_fee(mass_kg, hazardous=False):
    """Return the landing fee for one finite non-negative cargo mass."""
    if not isinstance(mass_kg, (int, float)):
        raise TypeError("mass must be numeric")
    if not math.isfinite(mass_kg) or mass_kg < 0:
        raise ValueError("mass must be finite and non-negative")
    if mass_kg <= 100:
        fee = 5
    elif mass_kg < 500:
        fee = 12
    else:
        fee = 25
    if hazardous:
        fee += 7
    return fee
```

PYTHON (CONTINUED)

```
def build_clearance(record, clock):
    """Build a new normalized clearance dictionary."""
    hazardous = bool(record.get("hazardous", False))
    return {
        "callsign": normalize_callsign(record["callsign"]),
        "mass_kg": record["mass_kg"],
        "hazardous": hazardous,
        "fee": landing_fee(record["mass_kg"], hazardous),
        "created_at": datetime.now(timezone.utc).isoformat(),
    }


def save_clearances(path, records):
    """Write every clearance as a UTF-8 JSON array and return its count."""
    records = list(records)
    path.write_text(
        json.dumps(records[:1], ensure_ascii=False, indent=2) + "\n",
        encoding="utf-8",
    )
    return len(records)


def load_clearances(path):
    """Load and return a clearance list from UTF-8 JSON."""
    return json.loads(path.read_text(encoding="utf-8"))


def build_parser():
    parser = argparse.ArgumentParser(prog="spaceport-dispatch")
    subparsers = parser.add_subparsers(dest="command", required=True)
    fee_parser = subparsers.add_parser("fee")
    fee_parser.add_argument("mass")
    fee_parser.add_argument("--hazardous", action="store_true")
    return parser


def main(argv=None, stdout=None, stderr=None):
    """Run the fee command and return a process-style status."""
    if stdout is None:
        stdout = sys.stdout
    if stderr is None:
        stderr = sys.stderr
    arguments = build_parser().parse_args(argv)
    try:
        mass = float(arguments.mass)
        fee = landing_fee(mass, arguments.hazardous)
    except (TypeError, ValueError) as error:
        stdout.write(f"spaceport-dispatch: {error}\n")
        return 2
    stdout.write(f"{fee}\n")
    return 0


if __name__ == "__main__":
    raise SystemExit(main())
```

Do not fix a suspicious line before a test exposes its violated rule. Begin with this scaffold:

PYTHON

```
from datetime import datetime, timezone
from io import StringIO

import pytest

from spaceport_dispatch import (
```

PYTHON (CONTINUED)

```
    build_clearance,
    landing_fee,
    load_clearances,
    main,
    normalize_callsign,
    save_clearances,
)


@pytest.fixture
def fixed_clock():
    def clock():
        return datetime(2042, 4, 5, 6, 7, tzinfo=timezone.utc)

    return clock


def test_ordinary_callsign_is_normalized():
    assert normalize_callsign("  nova   7 ") == "NOVA 7"


def test_light_non_hazardous_pod_costs_five():
    assert landing_fee(25) == 5


def test_main_prints_an_ordinary_fee():
    stdout = StringIO()
    stderr = StringIO()

    status = main(["fee", "25"], stdout, stderr)

    assert status == 0
    assert stdout.getvalue() == "5\n"
    assert stderr.getvalue() == ""
```

Run the baseline:

BASH

```
python -m pytest -q
```

These ordinary cases should pass. That does not clear the spaceport; it only confirms the project imports and three ordinary routes are ready for deeper inspection.

4. Inspect one subsystem at a time

Build the suite in this order:

1. callsign examples and empty input;
2. fee thresholds, hazardous addition, and invalid inputs;
3. fixed-clock clearance construction and input immutability;
4. manifest round trips with several records, Unicode, and an empty list;
5. direct command stdout/stderr/status behavior;
6. one child-process smoke test;
7. branch coverage questions; and
8. callsign properties and a named tab regression.

This order keeps the earliest red report narrow. If you add every test at once, six failures compete for attention and encourage changing several causes together.

Required public test names

Use these names where they apply so the progressive commands remain useful:

- `test_tab_is_collapsed_in_callsign_regression`
- `test_fee_boundaries`
- `test_hazardous_fee_adds_seven`
- `test_invalid_masses_are_rejected`
- `test_clearance_uses_supplied_clock_without_mutating_record`
- `test_manifest_round_trips_every_record`
- `test_invalid_mass_uses_stderr_and_status_two`
- `test_module_command_runs_as_a_process`
- `test_normalization_is_idempotent`
- `test_normalized_callsign_contains_only_single_spaces`

You may add descriptive helpers, fixtures, and test names. Do not rename production functions or broaden their signatures to avoid testing the contract.

5. Run progressive assertions

Check 1 — collect the baseline

```BASH
python -m pytest --collect-only -q
python -m pytest -q -x
```

Expected starting evidence: three collected tests pass.

Checks 2–6 — normalize callsigns

Create a five-row parameter table for ordinary spaces, repeated spaces, a tab, hyphen/digits, and outside whitespace. Add a separate empty-input exception test. Run:

```BASH
python -m pytest -q tests/test_spaceport_dispatch.py -k callsign -x
```

The tab or repeated-space node should fail against the starter. Preserve the smallest failing tab as the required regression before repairing the split.

Checks 7–20 — protect fees

Create seven base-fee rows including values just below, at, and just above 100 and 500. Add three hazardous rows and four invalid rows: `True`, `-0.1`, `float("inf")`, and `"heavy"`.

```BASH
python -m pytest -q tests/test_spaceport_dispatch.py -k fee -x
```

The `100` and Boolean cases should become separate red nodes. Confirm the compared fee or missing exception before editing production.

Check 21 — control the clock and preserve the caller’s record

Use `fixed_clock`. Copy the input record before the call. Assert the complete clearance, unchanged input, and exact timestamp. The starter should fail because its timestamp comes from the real clock.

```bash
python -m pytest -q \
  tests/test_spaceport_dispatch.py::test_clearance_uses_supplied_clock_without_mutating_record
```

Checks 22–24 — round-trip manifests

Use `tmp_path` to save and load two records including `"ÓRBITA-7"`; assert the returned count and complete list. Add an empty-list round trip and an invalid JSON exception test.

```bash
python -m pytest -q tests/test_spaceport_dispatch.py -k manifest -x
```

The two-record test should reveal that only one record was written.

Checks 25–27 — inspect the command boundary

Retain the ordinary direct test. Add the named invalid-mass test and one subprocess smoke test using `sys.executable -m spaceport_dispatch fee 100`.

```bash
python -m pytest -q tests/test_spaceport_dispatch.py -k "main or process" -x
```

The invalid direct test should show a diagnostic in stdout instead of stderr. The process test should use a valid boundary and expect status 0, stdout `"12\n"`, and empty stderr.

Checks 28–29 — search normalization properties

Define a bounded text strategy containing letters, digits, spaces, tabs, and hyphens. Add idempotence and allowed-whitespace properties. The no-single-space property may find the tab before your named regression does; retain both forms of evidence.

```bash
python -m pytest -q tests/test_spaceport_dispatch.py -k normalization
```

Inspect coverage without manufacturing a grade

```bash
python -m pytest \
  --cov=spaceport_dispatch \
  --cov-branch \
  --cov-report=term-missing
```

For every missing line or branch, write one of:

- public risk → add a focused assertion;
- already covered by a clearer boundary → explain why duplication adds little;
- unreachable after validation → consider simplifying production; or

- outside this challenge’s public contract → record the limit.

Do not change assertions merely to reach 100%.

Finish from a clean state

```bash
python -m pytest -q
python -m spaceport_dispatch fee 100
```

The suite should collect approximately 30 cases and pass. The real process should print `12` and exit successfully.

6. Use the hint ladder only when needed

Hint 1

For callsigns, no-argument `split()` recognizes runs of whitespace; `split(" ")` recognizes only ordinary spaces and retains empty pieces between repeated spaces. For fees, remember that `bool` is a subclass of `int`, so reject booleans before the ordinary numeric type check. Write each failing test before making either repair.

Hint 2

The fixed clock is a zero-argument callable. `build_clearance` should call it once and format that returned datetime. The manifest writer should serialize the complete `records` list, not a slice. Read the real temporary file back to prove what crossed the boundary.

Hint 3

For the invalid command, inspect `stdout.getvalue()`, `stderr.getvalue()`, and status independently. For the property strategy, generate the accepted callsign alphabet directly and assert that output contains neither tabs nor double spaces and equals its stripped form. Preserve the minimal tab example as a named regression.

7. Keep debugging evidence

Retain one record for a defect that was not the first one you noticed:

Field	Your evidence
Contract rule	What exact public promise applies?
Focused node	Which node or parameter ID did you run?
Red observation	Which phase, expression, and values appeared?
Hypothesis	Which single production cause explains them?
Controlled repair	Which one change tested the hypothesis?
Focused rerun	What changed in the same node?
Complete rerun	What did the approximately 30-case suite report?
Remaining limit	Which risk is still outside this suite?

Avoid entries such as “tests failed, fixed code.” A useful record could say:

`test_fee_boundaries[one-hundred-starts-middle-tier]` failed in the call phase: observed 5, expected 12. The contract makes 100 inclusive in the middle tier, while production uses `<= 100` for the light tier. Changing only that comparison to `< 100` made the node and complete fee table green.

8. Compare a complete inspection suite

Open this only after your own focused repairs. The solution is one design, not a reason to replace a working readable suite with identical formatting.

Show the complete test suite

PYTHON

```
import json
import subprocess
import sys
from datetime import datetime, timezone
from io import StringIO
from pathlib import Path

import pytest
from hypothesis import given
from hypothesis import strategies as st

from spaceport_dispatch import (
    build_clearance,
    landing_fee,
    load_clearances,
    main,
    normalize_callsign,
    save_clearances,
)


@pytest.fixture
def fixed_clock():
    def clock():
        return datetime(2042, 4, 5, 6, 7, tzinfo=timezone.utc)

    return clock


@pytest.mark.parametrize(
    ("raw", "expected"),
    [
        pytest.param("nova 7", "NOVA 7", id="ordinary-space"),
        pytest.param("nova   7", "NOVA 7", id="repeated-spaces"),
        pytest.param("nova\t7", "NOVA 7", id="tab"),
        pytest.param("lx-42", "LX-42", id="hyphen-and-digits"),
        pytest.param("  órbita 7 ", "ÓRBITA 7", id="outside-space"),
    ],
)
def test_callsign_examples(raw, expected):
    assert normalize_callsign(raw) == expected


def test_tab_is_collapsed_in_callsign_regression():
    assert normalize_callsign("0\t0") == "0 0"


def test_empty_callsign_is_rejected():
    with pytest.raises(ValueError, match="callsign must not be empty"):
        normalize_callsign(" \t ")
```

PYTHON (CONTINUED)

```
@pytest.mark.parametrize(
    ("mass_kg", "expected"),
    [
        pytest.param(0, 5, id="zero"),
        pytest.param(99.9, 5, id="below-one-hundred"),
        pytest.param(100, 12, id="one-hundred-starts-middle-tier"),
        pytest.param(100.1, 12, id="above-one-hundred"),
        pytest.param(499.9, 12, id="below-five-hundred"),
        pytest.param(500, 25, id="five-hundred-starts-heavy-tier"),
        pytest.param(900, 25, id="ordinary-heavy"),
    ],
)
def test_fee_boundaries(mass_kg, expected):
    assert landing_fee(mass_kg) == expected


@pytest.mark.parametrize(
    ("mass_kg", "expected"),
    [
        pytest.param(25, 12, id="light-plus-seven"),
        pytest.param(100, 19, id="middle-plus-seven"),
        pytest.param(500, 32, id="heavy-plus-seven"),
    ],
)
def test_hazardous_fee_adds_seven(mass_kg, expected):
    assert landing_fee(mass_kg, hazardous=True) == expected


@pytest.mark.parametrize(
    ("mass_kg", "error_type"),
    [
        pytest.param(True, TypeError, id="boolean"),
        pytest.param("heavy", TypeError, id="text"),
        pytest.param(-0.1, ValueError, id="negative"),
        pytest.param(float("inf"), ValueError, id="infinite"),
    ],
)
def test_invalid_masses_are_rejected(mass_kg, error_type):
    with pytest.raises(error_type):
        landing_fee(mass_kg)


def test_clearance_uses_supplied_clock_without_mutating_record(fixed_clock):
    record = {"callsign": " nova 7 ", "mass_kg": 100, "hazardous": True}
    original = record.copy()

    observed = build_clearance(record, fixed_clock)

    assert observed == {
        "callsign": "NOVA 7",
        "mass_kg": 100,
        "hazardous": True,
        "fee": 19,
        "created_at": "2042-04-05T06:07:00+00:00",
    }
    assert record == original


def test_manifest_round_trips_every_record(tmp_path):
    path = tmp_path / "clearances.json"
    records = [
        {"callsign": "NOVA 7", "fee": 12},
        {"callsign": "ÓRBITA-7", "fee": 25},
    ]

    count = save_clearances(path, records)
```

PYTHON (CONTINUED) (CONTINUED)

```
    assert count == 2
    assert load_clearances(path) == records
    assert "ÓRBITA-7" in path.read_text(encoding="utf-8")


def test_empty_manifest_round_trips(tmp_path):
    path = tmp_path / "empty.json"

    assert save_clearances(path, []) == 0
    assert load_clearances(path) == []


def test_invalid_manifest_preserves_json_error(tmp_path):
    path = tmp_path / "broken.json"
    path.write_text("{not-json", encoding="utf-8")

    with pytest.raises(json.JSONDecodeError):
        load_clearances(path)


def test_main_prints_an_ordinary_fee():
    stdout = StringIO()
    stderr = StringIO()

    status = main(["fee", "25"], stdout, stderr)

    assert status == 0
    assert stdout.getvalue() == "5\n"
    assert stderr.getvalue() == ""


def test_invalid_mass_uses_stderr_and_status_two():
    stdout = StringIO()
    stderr = StringIO()

    status = main(["fee", "nan"], stdout, stderr)

    assert status == 2
    assert stdout.getvalue() == ""
    assert stderr.getvalue() == (
        "spaceport-dispatch: mass must be finite and non-negative\n"
    )


def test_module_command_runs_as_a_process():
    project_root = Path(__file__).resolve().parents[1]
    result = subprocess.run(
        [sys.executable, "-m", "spaceport_dispatch", "fee", "100"],
        cwd=project_root,
        text=True,
        capture_output=True,
        check=False,
    )

    assert result.returncode == 0
    assert result.stdout == "12\n"
    assert result.stderr == ""


callsign_text = st.text(
    alphabet=st.characters(
        categories=("L", "N"),
        include_characters=" \t-",
    ),
    min_size=1,
    max_size=40,
).filter(lambda text: bool(text.split()))
```

PYTHON (CONTINUED) (CONTINUED) (CONTINUED)

```
@given(callsign_text)
def test_normalization_is_idempotent(text):
    once = normalize_callsign(text)
    assert normalize_callsign(once) == once


@given(callsign_text)
def test_normalized_callsign_contains_only_single_spaces(text):
    normalized = normalize_callsign(text)
    assert "\t" not in normalized
    assert "  " not in normalized
    assert normalized == normalized.strip()
```

The ordinary call example plus the listed parameter rows and property tests collects about 30 cases. If your process test runs from a different project shape, calculate its `cwd` deliberately; the goal is a fresh child process that can find the module without depending on state from a previous test.

Show the repaired application module

PYTHON

```
import argparse
import json
import math
import sys


def normalize_callsign(text):
    """Trim, collapse whitespace, and uppercase a non-empty callsign."""
    if not isinstance(text, str):
        raise TypeError("callsign must be text")
    normalized = " ".join(text.split()).upper()
    if not normalized:
        raise ValueError("callsign must not be empty")
    return normalized


def landing_fee(mass_kg, hazardous=False):
    """Return the landing fee for one finite non-negative cargo mass."""
    if isinstance(mass_kg, bool) or not isinstance(mass_kg, (int, float)):
        raise TypeError("mass must be numeric")
    if not math.isfinite(mass_kg) or mass_kg < 0:
        raise ValueError("mass must be finite and non-negative")
    if mass_kg < 100:
        fee = 5
    elif mass_kg < 500:
        fee = 12
    else:
        fee = 25
    if hazardous:
        fee += 7
    return fee


def build_clearance(record, clock):
    """Build a new normalized clearance dictionary."""
    hazardous = bool(record.get("hazardous", False))
    return {
        "callsign": normalize_callsign(record["callsign"]),
        "mass_kg": record["mass_kg"],
        "hazardous": hazardous,
        "fee": landing_fee(record["mass_kg"], hazardous),
        "created_at": clock().isoformat(),
    }
```

PYTHON (CONTINUED)

```
def save_clearances(path, records):
    """Write every clearance as a UTF-8 JSON array and return its count."""
    records = list(records)
    path.write_text(
        json.dumps(records, ensure_ascii=False, indent=2) + "\n",
        encoding="utf-8",
    )
    return len(records)


def load_clearances(path):
    """Load and return a clearance list from UTF-8 JSON."""
    return json.loads(path.read_text(encoding="utf-8"))


def build_parser():
    parser = argparse.ArgumentParser(prog="spaceport-dispatch")
    subparsers = parser.add_subparsers(dest="command", required=True)
    fee_parser = subparsers.add_parser("fee")
    fee_parser.add_argument("mass")
    fee_parser.add_argument("--hazardous", action="store_true")
    return parser


def main(argv=None, stdout=None, stderr=None):
    """Run the fee command and return a process-style status."""
    if stdout is None:
        stdout = sys.stdout
    if stderr is None:
        stderr = sys.stderr
    arguments = build_parser().parse_args(argv)
    try:
        mass = float(arguments.mass)
        fee = landing_fee(mass, arguments.hazardous)
    except (TypeError, ValueError) as error:
        stderr.write(f"spaceport-dispatch: {error}\n")
        return 2
    stdout.write(f"{fee}\n")
    return 0


if __name__ == "__main__":
    raise SystemExit(main())
```

9. Verify the changed requirement and process boundary

The launch inspector adds one rule after the first clean suite:

Cargo with mass exactly 1,000 kg receives a 4-credit heavy-lift surcharge, applied after the hazardous surcharge. Values below 1,000 are unchanged.

Use red–green–refactor:

1. add rows for `999.9` and `1000`, with and without hazardous cargo;
2. run the `1000` node against the current implementation and confirm the fee differs by exactly 4;
3. add the smallest clear production rule;
4. run the fee group, complete suite, and process command; and
5. inspect branch coverage for the new threshold.

Do not change old expected fees below 1,000. Do not mix the feature with a refactor until the changed contract is green.

Then launch from a fresh shell:

```bash
python -m spaceport_dispatch fee 1000 --hazardous
```

Expected result after the change is `36`: heavy base 25, hazardous 7, and heavy-lift surcharge 4.

10. Check your understanding

Clear the Buggy Spaceport for Launch readiness check

Choose the evidence that follows from the spaceport contract and suite.

1. Why must landing_fee reject True before the ordinary int/float type check?

- **A.** True is infinite
- **B.** Pytest cannot parametrize booleans
- **C.** Hazardous cargo cannot be Boolean
- **D.** bool is a subclass of int and would otherwise be accepted as mass 1

2. Which evidence best proves build_clearance used the supplied clock?

- **A.** The complete clearance contains the fixed clock’s exact ISO timestamp
- **B.** The real system time is within one second
- **C.** The test sleeps before asserting
- **D.** The function imports datetime

3. save_clearances returns 2 but writes only the first of two records. Which assertion exposes the defect most directly?

- **A.** Assert only that the path exists
- **B.** Load the file and compare the complete two-record list
- **C.** Assert the returned count is an int
- **D.** Mock json.dumps without reading output

4. An invalid mass returns status 2 but its message appears in stdout. Which contract is violated?

- **A.** Fee threshold selection
- **B.** Callsign normalization
- **C.** Diagnostics must use stderr while stdout remains data-only
- **D.** Manifest encoding

5. The repaired module reaches 100% statement coverage. What should the inspector conclude?

- **A.** All possible inputs are proven

- **B.** No property tests are needed
- **C.** Every future requirement is covered
- **D.** Every counted statement ran, but assertion quality, branches, and missing requirements still need review

Write your choices before checking the answer key at the end of this book.

11. Record the challenge result

Evidence rubric

Evidence	Ready to record when
Callsigns	Ordinary, repeated, tab, Unicode, hyphen, empty, regression, and property behavior pass.
Fees	Both thresholds, hazardous addition, Boolean/text/negative/infinite rejection, and the changed 1,000-kg rule pass.
Clearance	The exact injected time appears and the caller’s record remains unchanged.
Manifest	Several records, Unicode, empty data, and invalid JSON use a real isolated file boundary.
Command	Direct and child-process evidence preserve stdout, stderr, and status.
Coverage	Missing statements and branches are interpreted as risks or explicit limits, not chased as a grade.
Reproducibility	The focused groups and complete suite pass in a clean process.
Debugging	One record connects the contract, failed node, exact observation, one repair, and clean rerun.

This button stores a self-reported marker only in this browser. It does not submit work, grade the suite, verify identity, or issue a certificate.

Key points

- A testing challenge begins with a public contract and a reproducible red observation, not a hunt for secret bugs.
- Parametrized boundary IDs make fee failures precise without repetitive test bodies.
- `tmp_path`, explicit streams, a supplied clock, and one subprocess provide real evidence at different boundaries.
- Logic bugs remain valid Python; tests expose their observed behavior rather than syntax markers.
- A Hypothesis property explores a general whitespace rule while a named tab regression preserves the discovered history.

- Statement and branch coverage guide investigation but never replace assertions or justify changing correct expected values.
- Red–green–refactor adds the later 1,000-kg requirement without mixing it into the original repairs.
- A clean complete rerun and an evidence-rich debugging record make the suite explainable to the next inspector.

Continue learning

- pytest documentation: getting started and reading failures
- pytest documentation: fixtures and temporary test worlds
- pytest documentation: parametrization
- Coverage.py documentation: branch coverage
- Hypothesis documentation: generated examples and shrinking
- Next: Code Quality and Maintainability Overview

Unit 14: Code Quality and Maintainability

Online lesson and runnable notebook

Prepare a tested Python project for a sequence of readability, typing, automation, and behavior-preserving refactoring labs.

- **Level:** Python Foundations · Unit 14
- **Estimated time:** 24–34 hours including the challenge
- **Unit outcome:** Make Python projects easier to read and safer to change by combining human-readable code, useful types, automated checks, and small verified refactors.
- **Practice in:** A local Git project and terminal, with generated notebooks for focused code experiments

1. Two programs can return the same answer and still differ in quality

The Midnight Museum Quest awards points to teams that solve hidden clues. These two functions return the same score for the supplied attempt:

```python
def s(x):
    return x[0] * 30 + (25 if x[2] else 0) - max(0, x[1] - 45) * 2

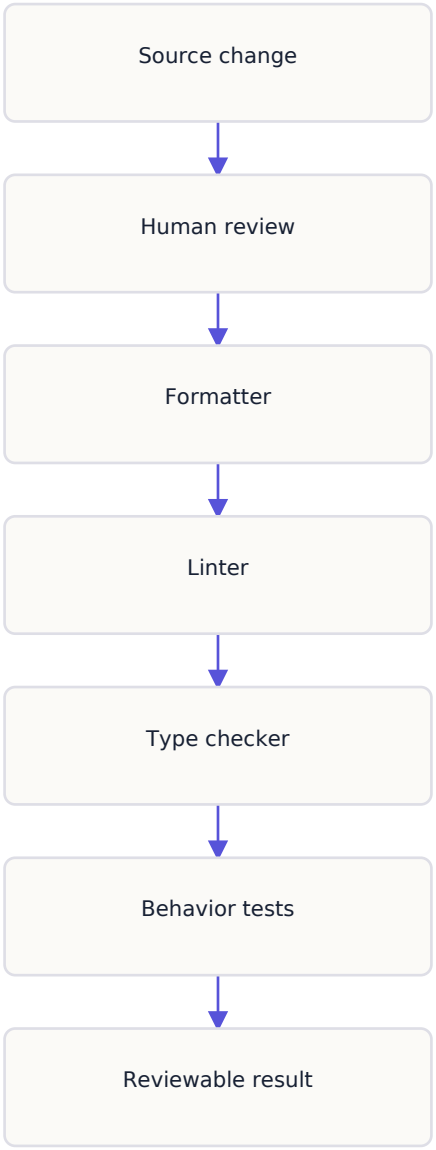

def score_attempt(clues_found, elapsed_minutes, found_bonus_clue):
    clue_points = clues_found * 30
    bonus_points = 25 if found_bonus_clue else 0
    overtime_minutes = max(0, elapsed_minutes - 45)
    overtime_penalty = overtime_minutes * 2
    return clue_points + bonus_points - overtime_penalty


assert s((6, 42, True)) == 205
assert score_attempt(6, 42, True) == 205
```

The first version is shorter. The second exposes the represented values, their units, and the stages of the rule. Now imagine that overtime becomes three points per minute, but only after minute 50. Which version can you verify more quickly? Which names would help you find every affected line? What behavior must stay unchanged for attempts completed before minute 50?

Code quality is not a beauty contest. It is the practical cost of answering questions like those accurately. In this unit, you will use human reading, static tools, tests, and small diffs as different kinds of evidence. No single check proves that software is correct or maintainable.

Notice that each check answers a different question before a change is accepted.



2. Questions this unit will help you answer

By the end of the unit, you should be able to answer these with code, command output, or a focused explanation:

1. Which names, boundaries, and intermediate values make a change safer to review?
2. What does an annotation communicate, and what still needs runtime validation?
3. How did MyPy or Ruff reach a diagnostic, and should the source or the configuration change?
4. Why did a pre-commit hook modify, skip, pass, or fail a file?
5. Can a clean CI runner reproduce the same quality result as your local project?
6. How can you improve structure one step at a time without silently changing observable behavior?

You will answer them while improving a small `src`-layout project called **Midnight Museum Quest**. Its tests begin green. Each lesson adds one kind of quality evidence without turning the project into a different application.

3. The learning sequence

Step	Lesson	What you will leave behind
1	Write Python People Can Follow	A readable scorer and a justification for choices no formatter can make
2	Make Contracts Visible with Type Annotations	Honest input, result, collection, and structured-record annotations
3	Read MyPy Errors and Close Type Gaps	An annotated diagnostic log and a strict clean run
4	Format and Lint a Project with Ruff	A reviewed Ruff patch and committed project policy
5	Catch Small Problems Before Commit	A local hook gate with modified-file and failure evidence
6	Repeat the Quality Gate in CI	A CI workflow that repeats the local commands on a clean runner
7	Refactor in Small, Verified Steps	A sequence of small diffs protected by the complete gate
8	Unit Challenge	A maintainable Clockwork Observatory project and its quality record

The order matters. Readability gives you a vocabulary for intent. Annotations make selected contracts checkable. Ruff and MyPy turn committed policy into repeatable feedback. Hooks shorten the feedback loop; CI checks committed state on a clean machine. Only then do you use the whole gate to support a larger refactor.

4. Prepare the local project

Unit 10 introduced environments, `pyproject.toml`, `src/` layouts, and Git. Unit 13 introduced pytest and behavior baselines. Bring those skills forward; this unit applies them rather than reteaching every command.

Use this shape for the connected project:

```text
midnight-museum/
├── pyproject.toml
├── .pre-commit-config.yaml
├── .github/
│   └── workflows/
│       └── quality.yml
├── src/
│   └── midnight_museum/
│       ├── __init__.py
│       └── quest.py
└── tests/
    └── test_quest.py
```

From the project root, confirm the tools that are already declared by the course repository or install them in your own development environment:

```bash
python --version
ruff --version
mypy --version
pre-commit --version
pytest --version
```

The course examples are verified with Python 3.10-compatible syntax, Ruff 0.15.20, MyPy 2.2.0, pre-commit 4.6.0, and pytest 9.1.1. A later compatible version may format or phrase a diagnostic differently. When output differs, compare the rule code and behavior instead of forcing your terminal to match a screenshot character for character.

Before changing style or types, establish the behavior baseline:

```bash
pytest -q
```

```text
3 passed in 0.01s
```

Start green, then keep the evidence

A green baseline does not prove the program has no defects. It proves that the current tests pass in the current environment. Save the command, result, and commit state so a later failure can be attributed to a controlled change. :::

5. Use notebooks and the project for different jobs

The generated notebooks are useful for comparing two functions, inspecting annotations, tracing a type narrowing branch, and experimenting with small refactors. They are not a substitute for a Git working tree:

- pre-commit installs scripts under `.git/hooks/`;
- Ruff and MyPy discover committed project configuration from the filesystem;
- CI reads a workflow from `.github/workflows/`; and
- a clean runner is valuable precisely because it does not inherit notebook state.

Complete the project-shaped labs in a local directory. If a managed learning environment prevents GitHub access, you can still author the workflow and investigate the supplied CI transcripts; pushing a repository is not required to understand the gate.

6. Plan realistic stopping points

This unit involves more active setup and diagnosis than a syntax lesson. A reasonable rhythm is:

- **Session 1:** overview and readable Python;
- **Session 2:** type annotations;
- **Session 3:** MyPy diagnostics;
- **Session 4:** Ruff configuration and repair;

- **Session 5:** pre-commit hooks;
- **Session 6:** CI workflows;
- **Sessions 7–8:** refactoring lab and challenge.

Stop after a clean checkpoint rather than halfway through an unexplained tool failure. Keep an evidence log with these columns:

Command or change	Expected result	Observed result	Explanation or next step
`pytest -q` before edits	Existing suite passes	3 passed	Baseline is usable
One naming change	Same test result	…	…
`mypy src`	No type errors	…	…
Complete local gate	Every command exits zero	…	…

7. Begin with the human reader

Automated tools are fast, but they cannot decide whether `m` means minutes, meters, a museum, or a mysterious temporary value. Start by learning which choices belong to the maintainer and how to justify them with the next likely change.

Start Lesson 1: Write Python People Can Follow

Write Python People Can Follow

Online lesson and runnable notebook

Improve names, layout, expressions, and function structure so a reader can verify a change without translating the program first.

- **Level:** Beginner project maintainer
- **Estimated time:** 3–4 hours
- **You will learn:** Make readability choices that expose represented values, units, control flow, and the next likely change while preserving behavior.
- **Practice in:** The Midnight Museum project, a notebook for comparisons, and a local test runner

1. Readability reduces the work before the real work

A Midnight Museum team earns 30 points per clue, 25 points for a bonus clue, and loses 2 points for every minute after minute 45. You need to change the bonus from 25 to 40. Start with this valid implementation:

```python
def calc(x):
    a = x[0] * 30
    b = 25 if x[2] else 0
    c = max(0, x[1] - 45) * 2
    return a + b - c


assert calc((6, 42, True)) == 205
```

Before editing, answer four questions:

1. Which position contains elapsed minutes?
2. Does `b` represent the presence of a bonus, the clue, or the awarded points?
3. Can the result become negative?
4. Which line must change, and how will you prove that ordinary attempts did not change accidentally?

The interpreter has enough information. A maintainer does not. The tuple positions and short names force the reader to reconstruct meaning before reasoning about the change.

Rewrite only the representation and names:

```python
def score_attempt(clues_found, elapsed_minutes, found_bonus_clue):
    clue_points = clues_found * 30
    bonus_points = 25 if found_bonus_clue else 0
    overtime_minutes = max(0, elapsed_minutes - 45)
    overtime_penalty = overtime_minutes * 2
    return clue_points + bonus_points - overtime_penalty


assert score_attempt(6, 42, True) == 205
```

The second function is longer, but each line answers a domain question. A future change to overtime has a visible location. Length is not the goal; reducing unnecessary reconstruction is.

Name the value and its unit

A strong name usually combines the represented idea with information needed to use it safely:

Name	What the reader still has to discover
`t`	Everything
`time`	Elapsed time, start time, deadline, or clock time? Which unit?
`elapsed`	Which unit?
`elapsed_minutes`	The represented duration and unit are visible

Units matter whenever two values of the same Python type are not interchangeable. `elapsed_minutes`, `distance_km`, and `price_cents` make a wrong calculation easier to spot even before the type lesson.

Longer is not automatically clearer

`number_of_clues_found_by_the_current_team` repeats context already supplied by `score_attempt`. Prefer the shortest name that stays unambiguous in its scope. :::

2. Let conventions carry familiar information

PEP 8 gives Python projects a shared set of reading conventions. It is guidance for consistency, not a law that overrides clarity. PEP 8 itself says consistency within a project and within a function matters greatly, and that a recommendation can be ignored when following it would reduce readability.

Common naming shapes communicate a role before the reader opens the definition:

```python
MAX_CLUES = 8
OVERTIME_START_MINUTES = 45


def score_attempt(clues_found, elapsed_minutes, found_bonus_clue):
    ...


class QuestResult:
    ...


def _validate_attempt(clues_found, elapsed_minutes):
    ...
```

- `lowercase_with_underscores` is conventional for functions and variables.
- `CapitalizedWords` identifies classes.
- `UPPER_CASE_WITH_UNDERSCORES` signals a module-level value treated as a constant by the project.
- One leading underscore marks an implementation detail such as `_validate_attempt`; it does not create security or runtime privacy.

- Avoid inventing `__double_underscore__` names. Python reserves that shape for documented language protocols.

A trailing underscore avoids a keyword collision without an obscure abbreviation:

```python
def group_by_class(class_):
    return f"Gallery class: {class_}"


assert group_by_class("clockwork") == "Gallery class: clockwork"
```

Make booleans read as questions

Compare the branch conditions:

```python
def bonus_points(bonus):
    return 25 if bonus else 0


def bonus_points(found_bonus_clue):
    return 25 if found_bonus_clue else 0
```

`if found_bonus_clue` can be read aloud as a yes/no statement. Names beginning with `is_`, `has_`, `can_`, or a past-tense action often help, but choose the wording that matches the domain. Avoid double negatives:

```python
def can_enter_gallery(has_ticket, is_gallery_closed):
    return has_ticket and not is_gallery_closed


assert can_enter_gallery(True, False) is True
```

`not is_not_open` would make the reader perform an avoidable logic conversion.

Checkpoint: names and conventions

Names that carry meaning

Choose the option that gives a future reader the strongest evidence.

1. A function receives a duration used in a per-minute penalty. Which parameter is clearest?

- **A.** elapsed_minutes
- **B.** value
- **C.** elapsed
- **D.** e

2. Which name follows the usual Python convention for a module-level overtime threshold treated as constant?

- **A.** OvertimeStart
- **B.** OVERTIME_START_MINUTES

- **C.** overtimeStart
- **D. overtime**

3. Which condition communicates its yes/no meaning most directly?

- **A.** if b:
- **B.** if not not_open:
- **C.** if found_bonus_clue:
- **D.** if value == True:

Write your choices before checking the answer key at the end of this book.

3. Shape the file so its groups are visible

Whitespace does not repair poor design, but consistent layout helps readers see where one idea ends and another begins. Put imports at the top, group standard library, third-party, and local imports, and keep one import per line unless a `from` import is clearer:

```python
from dataclasses import dataclass
from pathlib import Path

import pytest

from midnight_museum.scoring import score_attempt
```

In an ordinary module, separate top-level classes and functions with two blank lines. Within a function, use a blank line to separate meaningful stages—not after every statement.

Long expressions are safest to wrap inside parentheses, brackets, or braces. The delimiters make continuation explicit and let every indentation level remain four spaces:

```python
leaderboard_line = (
    f"{position}. {team_name} — "
    f"{score} points in {elapsed_minutes} minutes"
)
```

A trailing comma makes multi-line collections and calls easier to extend and produces smaller diffs:

```python
quality_commands = [
    "ruff check src tests",
    "mypy src",
    "pytest -q",
]
```

Do not compress independent actions onto one physical line:

```python
def record_score(scores, team, score):
    scores[team] = score
    return scores[team]


scores = {}
assert record_score(scores, "Moon Moths", 205) == 205
```

The equivalent `scores[team] = score; return scores[team]` is valid Python, but it hides two actions on a line and produces a worse location when one action fails.

Formatting exposes structure; it does not choose structure

A formatter can normalize spaces, wrapping, quotes, and trailing commas. It cannot decide that `x[1]` should become `elapsed_minutes`, that a tuple should be a structured record, or that one function owns too many responsibilities. Those choices require domain knowledge. Lesson 4 will show exactly what Ruff can and cannot automate.

4. Keep the main path easy to find

Nested branches make a reader hold unfinished conditions in memory:

PYTHON

```
def admission_message(has_ticket, is_gallery_open, age):
    if has_ticket:
        if is_gallery_open:
            if age >= 12:
                return "Enter the puzzle gallery"
            return "Enter with an adult"
        return "The gallery is closed"
    return "A ticket is required"
```

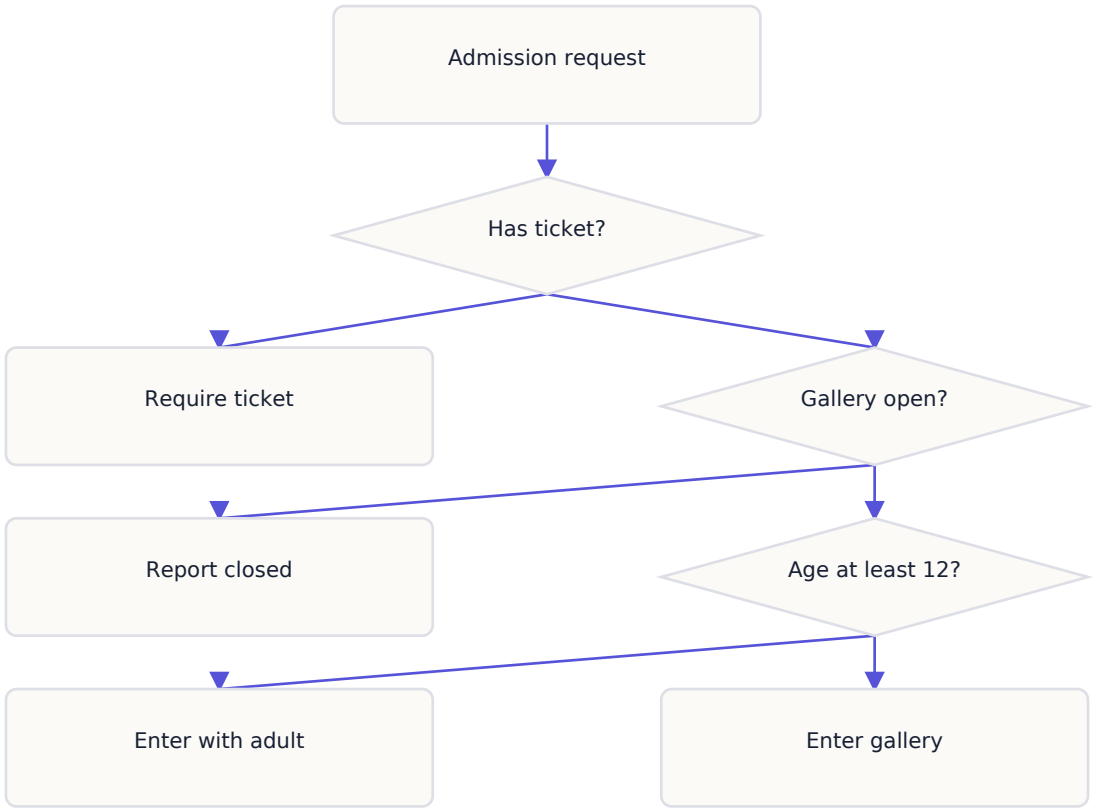
Guard clauses handle rejected cases first and leave the ordinary successful path at the lowest indentation level:

PYTHON

```
def admission_message(has_ticket, is_gallery_open, age):
    if not has_ticket:
        return "A ticket is required"
    if not is_gallery_open:
        return "The gallery is closed"
    if age < 12:
        return "Enter with an adult"
    return "Enter the puzzle gallery"
```

Both functions can be correct. The second is helpful when the rejected cases are independent and the final path is the main action. Guard clauses are not a rule to apply blindly: a small two-sided `if/else` can remain clearer when both branches have equal importance.

The guard-clause version closes rejected paths before the main result.



Has ticket? → Require ticket: No

Has ticket? → Gallery open?: Yes

Gallery open? → Report closed: No

Gallery open? → Age at least 12?: Yes

Age at least 12? → Enter with adult: No

Age at least 12? → Enter gallery: Yes

Keep one level of detail in view

This function mixes scoring rules, sorting, and text presentation:

PYTHON

```
def leaderboard(attempts):
    rows = []
    for attempt in attempts:
        score = attempt["clues_found"] * 30
        if attempt["bonus_clue"]:
            score += 25
        score -= max(0, attempt["elapsed_minutes"] - 45) * 2
        rows.append((score, attempt["team"]))
    rows.sort(reverse=True)
    return "\n".join(f"{team}: {score}" for score, team in rows)
```

Do not split it merely to maximize the number of functions. Split when the parts have distinct reasons to change and useful names:

PYTHON

```
def score_attempt(attempt):
    clue_points = attempt["clues_found"] * 30
```

PYTHON (CONTINUED)

```
    bonus_points = 25 if attempt["bonus_clue"] else 0
    overtime_penalty = max(0, attempt["elapsed_minutes"] - 45) * 2
    return clue_points + bonus_points - overtime_penalty


def rank_attempts(attempts):
    scored = [(score_attempt(attempt), attempt["team"]) for attempt in attempts]
    return sorted(scored, reverse=True)


def render_leaderboard(attempts):
    return "\n".join(
        f"{team}: {score}" for score, team in rank_attempts(attempts)
    )
```

Now a scoring change has a named home, and presentation can change without editing the calculation. Later lessons will refine the record types and tie rule.

Checkpoint: layout and control flow

Layout that reveals the program

Choose the answer that best supports the stated maintenance task.

1. Which improvement requires domain knowledge rather than an automatic formatter?

- **A.** Normalize spaces around an operator
- **B.** Wrap a long call
- **C.** Add a trailing comma
- **D.** Rename `t` to `elapsed_minutes` because the penalty is per minute

2. Why can independent guard clauses make `admission_message` easier to change?

- **A.** They close rejected cases early and leave the successful path visible
- **B.** They make every function execute faster
- **C.** They remove the need for tests
- **D.** They force all branches to return the same text

3. What is the strongest reason to separate scoring from leaderboard rendering?

- **A.** Every function must contain exactly three lines
- **B.** The calculation and text format have different reasons to change
- **C.** Python cannot sort values inside a rendering function
- **D.** PEP 8 forbids loops and strings in one function

Write your choices before checking the answer key at the end of this book.

5. Prefer clarity over clever compression

A concise expression can be excellent when its shape is familiar:

```python
active_teams = [
    attempt["team"]
    for attempt in attempts
    if attempt["clues_found"] > 0
]
```

Compression becomes expensive when it hides several rules:

```python
scores = {
    attempt["team"]: max(
        0,
        attempt["clues_found"] * 30
        + (25 if attempt["bonus_clue"] else 0)
        - max(0, attempt["elapsed_minutes"] - 45) * 2,
    )
    for attempt in attempts
    if attempt["team"].strip()
}
```

The expression is valid, but validation, scoring, filtering, and construction compete for attention. A loop with named stages makes debugging easier:

```python
scores = {}
for attempt in attempts:
    team_name = attempt["team"].strip()
    if not team_name:
        continue

    score = score_attempt(attempt)
    scores[team_name] = max(0, score)
```

Choose based on what the next reader must verify. A comprehension with one mapping and one filter is often clear. Several nested conditions, side effects, or independent domain rules usually deserve statements and names.

Comments should add missing reasons

This comment merely translates syntax:

```python
# Add 25 to score.
score += 25
```

This one preserves information that the expression cannot show:

```python
# The paper-map clue stays worth 25 points to match the printed rule cards.
score += 25
```

A stale comment is worse than no comment because it sends the maintainer toward a false contract. Keep the rule in tests or project documentation as well when it is public behavior. Unit 15 will teach documentation design in depth.

6. Review a change by behavior and readability

Suppose the requirement changes to “overtime begins after 50 minutes and costs 3 points per minute.” A controlled patch should:

1. update or add behavior examples at minutes 50 and 51;
2. rename a misleading threshold if necessary;
3. change the threshold and rate in one visible calculation;
4. rerun the focused tests; and
5. inspect the diff for an unrelated style rewrite.

```python
OVERTIME_START_MINUTES = 50
OVERTIME_PENALTY_PER_MINUTE = 3


def overtime_penalty(elapsed_minutes):
    overtime_minutes = max(0, elapsed_minutes - OVERTIME_START_MINUTES)
    return overtime_minutes * OVERTIME_PENALTY_PER_MINUTE


assert overtime_penalty(50) == 0
assert overtime_penalty(51) == 3
```

The names make both axes of the change explicit. The assertions establish the threshold boundary. A later type annotation will show that the values are integers, but only these names show that they are minutes and points.

Checkpoint: readable change evidence

A change another maintainer can verify

Use the requirement and code evidence, not line count, to choose.

1. When is a loop preferable to the dense score comprehension shown above?

- **A.** Only when the collection has more than ten items
- **B.** Whenever a formatter changes quotes
- **C.** When several rules and decisions need names or separate debugging evidence
- **D.** Because comprehensions are not Python

2. Which comment adds information that is not already visible in `score += 25`?

- **A.** Add points
- **B.** Increase score by twenty-five
- **C.** Use plus equals
- **D.** Keep 25 points to match the printed rule cards

3. Before changing the overtime threshold from 45 to 50, what is the strongest first evidence to add?

- **A.** Checks for minutes 50 and 51 with the expected penalties

- **B.** A screenshot of the formatted source
- **C.** A shorter function name
- **D.** An assertion that every score is an integer

Write your choices before checking the answer key at the end of this book.

7. Lab: make the museum scorer reviewable

Start from this behavior-correct but difficult function:

```python
def make_board(xs):
    out = []
    for x in xs:
        n = x["c"] * 30 + (25 if x["b"] else 0) - max(0, x["m"] - 45) * 2
        out.append((max(0, n), x["t"]))
    out.sort(reverse=True)
    return "\n".join(f"{t}: {n}" for n, t in out)
```

Use these acceptance checks before and after every controlled change:

```python
attempts = [
    {"t": "Moon Moths", "c": 6, "m": 42, "b": True},
    {"t": "Brass Bats", "c": 7, "m": 50, "b": False},
]

expected = "Moon Moths: 205\nBrass Bats: 200"
assert make_board(attempts) == expected
assert make_board([]) == ""
```

Complete these stages:

1. Write a table explaining every current key and local name.
2. Replace vague names with domain names without changing the record keys yet.
3. Extract a scoring function whose intermediate values reveal the rule.
4. Extract rendering only if its responsibility becomes clearer.
5. Decide whether `reverse=True` expresses the tie behavior you want. Do not change the public behavior during this lab; record the concern for a future requirement.
6. Run the assertions after each stage and keep one before/after diff.
7. Explain three decisions that a formatter could not make for you.

Hint A: map the current keys

`t` is the team name, `c` is clues found, `m` is elapsed minutes, and `b` records whether the team found the bonus clue. Rename locals first; changing keys and structure simultaneously makes a failure harder to locate.

Hint B: extract the calculation

A useful extracted function can name `clue_points`, `bonus_points`, `overtime_minutes`, and `overtime_penalty`. Keep the final `max(0, ...)` because that is current observable behavior.

Hint C: inspect the tie rule

Sorting `(score, team)` tuples with `reverse=True` orders both score and team in descending order. Preserve it for this behavior-preserving lab, but write down whether an alphabetical ascending tie rule would be more understandable as a future behavior change.

Show one complete behavior-preserving refactor

PYTHON

```
def score_attempt(attempt):
    clue_points = attempt["c"] * 30
    bonus_points = 25 if attempt["b"] else 0
    overtime_minutes = max(0, attempt["m"] - 45)
    overtime_penalty = overtime_minutes * 2
    return max(0, clue_points + bonus_points - overtime_penalty)


def make_board(attempts):
    ranked_scores = []
    for attempt in attempts:
        team_name = attempt["t"]
        score = score_attempt(attempt)
        ranked_scores.append((score, team_name))

    ranked_scores.sort(reverse=True)
    return "\n".join(
        f"{team_name}: {score}" for score, team_name in ranked_scores
    )


attempts = [
    {"t": "Moon Moths", "c": 6, "m": 42, "b": True},
    {"t": "Brass Bats", "c": 7, "m": 50, "b": False},
]
assert make_board(attempts) == "Moon Moths: 205\nBrass Bats: 200"
assert make_board([]) == ""
```

This version keeps the public function name, input keys, output, zero floor, and descending tuple tie rule. Changing the data shape or tie policy would be a separate behavior/interface change with its own tests.

Key points

- Readability is the effort required to verify meaning and make a safe change, not the fewest lines.
- Names should expose domain roles and units without repeating obvious scope.
- PEP 8 supplies shared conventions, but project consistency and clarity take priority over mechanical obedience.
- Layout, guard clauses, and cohesive functions can reveal structure; a formatter cannot choose the right domain structure for you.
- Prefer a clear loop when compression hides several rules or debugging points.
- Protect behavior before renaming or restructuring, and keep each diff focused.

References

- PEP 8: Style Guide for Python Code
- PEP 20: The Zen of Python
- Software Carpentry: Programming Style

- Python tutorial: Coding Style

Continue to Lesson 2: Make Contracts Visible with Type Annotations

Make Contracts Visible with Type Annotations

Online lesson and runnable notebook

Annotate function boundaries, collections, optional results, and structured records without confusing static information with runtime validation.

- **Level:** Python Foundations
- **Estimated time:** 4–5 hours
- **You will learn:** Express honest, useful type contracts for real project boundaries while keeping runtime validation where external values enter.
- **Practice in:** A notebook for type shapes and the Midnight Museum local project

1. An annotation makes an agreement visible

The readable scorer from Lesson 1 still leaves some questions unanswered. Can `clues_found` be text? Can the function return `None`? Is `found_bonus_clue` really a boolean or a string such as `"yes"`?

Annotations place those expectations in the function signature:

PYTHON

```
def score_attempt(
    clues_found: int,
    elapsed_minutes: int,
    found_bonus_clue: bool,
) -> int:
    clue_points = clues_found * 30
    bonus_points = 25 if found_bonus_clue else 0
    overtime_minutes = max(0, elapsed_minutes - 45)
    return max(0, clue_points + bonus_points - overtime_minutes * 2)


assert score_attempt(6, 42, True) == 205
```

Before continuing, answer these questions:

1. Which values cross the function boundary?
2. What value leaves after a successful call?
3. Does `: int` reject a negative integer?
4. Does Python automatically convert `"6"` to `6`?
5. Which tool can compare callers with this declared contract before runtime?

The signature answers the first two. It does **not** enforce the domain rules or convert input. Python stores annotations as metadata. A static checker, editor, or documentation tool may inspect them, but an ordinary call still receives the objects supplied by the caller.

PYTHON

```
def repeat_clue(clue: str, times: int) -> str:
    return clue * times
```

PYTHON (CONTINUED)

```
assert repeat_clue("★", 3) == "★★★"
```

This call is allowed to begin at runtime even though it conflicts with the annotation:

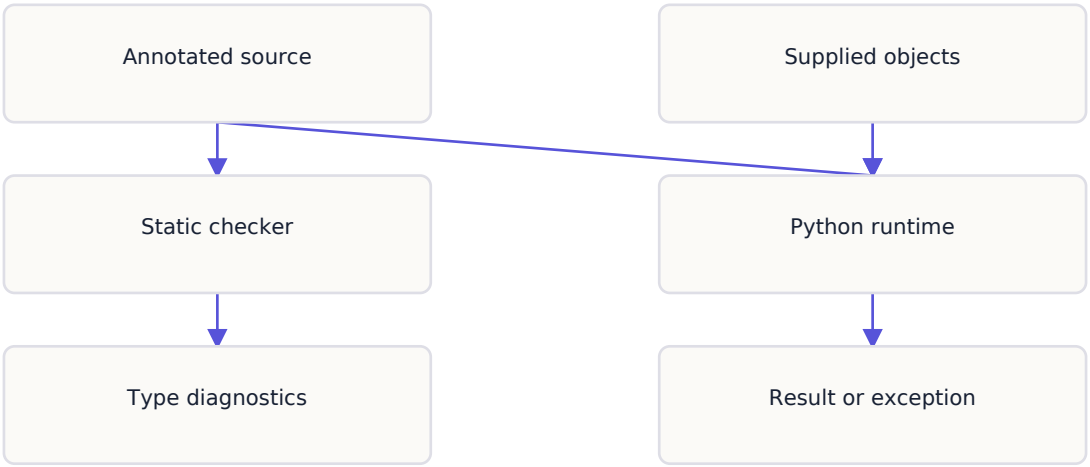
PYTHON

```
def repeat_clue(clue: str, times: int) -> str:
    return clue * times


bad_times = "3"
```

Calling `repeat_clue("★", bad_times)` would raise `TypeError` at multiplication; the annotation itself does not intercept the call. MyPy can report the mismatch without executing it. The next lesson teaches that diagnostic workflow.

Annotations are inspected by static tools, while runtime execution still uses the supplied objects.



2. Annotate boundaries, not every obvious temporary value

Parameters and returned values are high-value annotation sites because other code depends on them. A function that performs an action and returns no useful value should state `-> None`:

PYTHON

```
def announce_team(team_name: str) -> None:
    print(f"Next team: {team_name}")
```

Local values are usually inferred from their expressions:

PYTHON

```
def overtime_penalty(elapsed_minutes: int) -> int:
    overtime_minutes = max(0, elapsed_minutes - 45)
    penalty = overtime_minutes * 2
    return penalty
```

A checker can infer that `overtime_minutes` and `penalty` are integers. Adding `: int` to every local repeats information and can make the important boundary harder to see.

Annotate a local when it declares an initially empty collection or resolves an ambiguity:

```python
def collect_team_names(records: list[dict[str, object]]) -> list[str]:
    team_names: list[str] = []
    for record in records:
        team = record.get("team")
        if isinstance(team, str):
            team_names.append(team)
    return team_names
```

Without `list[str]`, an empty list has no element from which a checker can infer the intended item type at that point.

Built-in collections describe their contents

Python 3.10 supports the built-in generic forms used by this course:

```python
team_names: list[str] = ["Moon Moths", "Brass Bats"]
score_by_team: dict[str, int] = {"Moon Moths": 205}
position: tuple[int, str] = (1, "Moon Moths")
coordinates: tuple[int, ...] = (4, 9, 12)
unique_galleries: set[str] = {"Atrium", "Clock Hall"}
```

Read `dict[str, int]` as “a dictionary whose keys are strings and whose values are integers.” `tuple[int, str]` describes two fixed positions with different types; `tuple[int, ...]` describes any number of integer positions.

Checkpoint: annotations and runtime

What annotations do—and do not do

Choose the statement supported by ordinary Python runtime behavior.

1. What does `times: int` do during an ordinary call to `repeat_clue`?

- **A.** It converts any string to an integer
- **B.** It records expected type information but does not automatically validate or convert the argument
- **C.** It prevents every TypeError
- **D.** It runs MyPy before the function body

2. Which return annotation best describes a function that prints an announcement and intentionally returns no useful value?

- **A.** -> str
- **B.** -> bool
- **C.** -> None
- **D.** No annotation, because printing cannot be typed

3. Why is `team_names: list[str] = []` a useful local annotation?

- **A.** It makes append run faster
- **B.** It makes the list immutable

- **C.** It changes the empty list into a tuple
- **D.** It declares the intended element type before any element exists to support inference

Write your choices before checking the answer key at the end of this book.

3. Say when a result can be absent

A search might find a matching team or find nothing. Do not annotate it as `str` if `None` is a real result:

```python
def first_team_over(
    scores: dict[str, int],
    minimum_score: int,
) -> str | None:
    for team_name, score in scores.items():
        if score >= minimum_score:
            return team_name
    return None
```

`str | None` is a union: the result may be either type. The caller must separate those cases before using string operations:

```python
winner = first_team_over({"Moon Moths": 205}, 200)
if winner is None:
    message = "No team reached the threshold"
else:
    message = winner.upper()

assert message == "MOON MOTHS"
```

The `is None` branch does more than prevent an exception. It narrows the value: a static checker can treat `winner` as `str` inside the `else` block.

Do not use a falsey check when empty text is a valid, distinct result:

```python
def display_search_result(result: str | None) -> str:
    if result is None:
        return "No match"
    if result == "":
        return "The matching label is empty"
    return result
```

`if not result` would combine `None` and `""` even though the contract gives them different meanings.

Unions should represent real cases

`str | int | float | list[str] | None` may be honest for a raw external value, but it is burdensome inside the core of a program. Validate or convert at the boundary so deeper functions receive a smaller, more useful type.

```python
def parse_clue_count(raw_value: object) -> int:
    if isinstance(raw_value, bool):
        raise ValueError("clue count cannot be boolean")
    if isinstance(raw_value, int):
        return raw_value
    if isinstance(raw_value, str) and raw_value.isdigit():
```

PYTHON (CONTINUED)

```
        return int(raw_value)
    raise ValueError("clue count must be an integer")


assert parse_clue_count("6") == 6
assert parse_clue_count(6) == 6
```

`object` says the boundary may receive any Python object, but only operations valid for all objects are initially allowed. The `isinstance` branches both validate at runtime and narrow the type for static analysis.

4. Choose the smallest honest collection interface

A parameter annotated `list[Attempt]` promises that the implementation may need list-specific, mutable behavior. If the function only loops, that contract is unnecessarily narrow.

PYTHON

```
from collections.abc import Iterable


def total_scores(scores: Iterable[int]) -> int:
    return sum(scores)


assert total_scores([205, 200]) == 405
assert total_scores((205, 200)) == 405
assert total_scores(score for score in [205, 200]) == 405
```

`Iterable[int]` accepts anything that can produce integers one at a time, including a generator. That flexibility creates a responsibility: an iterable may be single-use. This implementation is wrong for a generator because it tries to traverse twice:

PYTHON

```
def average_score(scores: Iterable[int]) -> float | None:
    values = list(scores)
    if not values:
        return None
    return sum(values) / len(values)
```

Materializing once makes the intended two operations honest. For other needs:

- `Collection[T]` supports iteration, `len()`, and membership;
- `Sequence[T]` adds stable integer indexing and order;
- `Mapping[K, V]` supports read-only key lookup;
- `MutableSequence[T]` or `list[T]` is appropriate when mutation is part of the contract.

PYTHON

```
from collections.abc import Mapping, Sequence


def team_score(scores: Mapping[str, int], team_name: str) -> int:
    return scores[team_name]


def first_team(team_names: Sequence[str]) -> str | None:
    if not team_names:
        return None
```

PYTHON (CONTINUED)

```
    return team_names[0]
```

Choose from the operations the function actually needs, not from the concrete object used by the initial caller.

Checkpoint: absence and collection contracts

Honest result and collection shapes

Match each annotation to the behavior the function actually needs.

1. A search returns a team name or no match. Which annotation states both outcomes?

- **A.** -> str | None
- **B.** -> str
- **C.** -> object
- **D.** -> list[str]

2. A function only loops once over integer scores. Which input contract is the least restrictive honest choice?

- **A.** list[int]
- **B.** Iterable[int]
- **C.** dict[str, int]
- **D.** tuple[int, int]

3. Why can counting an `Iterable[int]` and then summing it in a second loop fail?

- **A.** Integers cannot be iterated
- **B.** Every iterable is unordered
- **C.** A generator may be exhausted by the first traversal
- **D.** Type annotations remove values after use

Write your choices before checking the answer key at the end of this book.

5. Give structured records a reusable shape

Repeated `dict[str, object]` annotations say little about available keys. A `TypedDict` describes a dictionary record while leaving its runtime value as an ordinary dictionary:

PYTHON

```
from typing import TypedDict


class Attempt(TypedDict):
    team: str
    clues_found: int
    elapsed_minutes: int
    bonus_clue: bool


def score_attempt(attempt: Attempt) -> int:
```

```
PYTHON (CONTINUED)
    clue_points = attempt["clues_found"] * 30
    bonus_points = 25 if attempt["bonus_clue"] else 0
    overtime_penalty = max(0, attempt["elapsed_minutes"] - 45) * 2
    return max(0, clue_points + bonus_points - overtime_penalty)


museum_attempt: Attempt = {
    "team": "Moon Moths",
    "clues_found": 6,
    "elapsed_minutes": 42,
    "bonus_clue": True,
}
assert score_attempt(museum_attempt) == 205
```

A checker can now report a missing `team` key or a string assigned to `elapsed_minutes`. At runtime, `museum_attempt` remains `dict`:

```
PYTHON
assert isinstance(museum_attempt, dict)
```

Use a dataclass when the record is a Python object with constructor behavior, methods, equality, or controlled mutability:

```
PYTHON
from dataclasses import dataclass


@dataclass(frozen=True)
class QuestResult:
    team: str
    score: int
    elapsed_minutes: int


result = QuestResult("Moon Moths", 205, 42)
assert result.team == "Moon Moths"
```

`TypedDict` is often natural near JSON-shaped dictionaries. A dataclass is usually clearer for a core domain object. Neither validates an untrusted JSON file automatically; Unit 9’s boundary validation still applies.

Use an alias to name a repeated domain type

Python 3.10-compatible aliases can use `TypeAlias`:

```
PYTHON
from typing import Literal, TypeAlias

Decision: TypeAlias = Literal["winner", "runner-up", "participant"]
ScoreByTeam: TypeAlias = dict[str, int]


def decision_for(position: int) -> Decision:
    if position == 1:
        return "winner"
    if position == 2:
        return "runner-up"
    return "participant"
```

`Literal` is useful when a value really is limited to a small closed vocabulary. Do not list hundreds of dynamic strings as literals. A class, enum, or validated string may fit better depending on the domain.

6. Keep external validation and static checking together

Suppose a JSON record enters as `dict[str, object]`. An annotation claiming it is already `Attempt` would make the checker quiet by lying about the boundary. Validate first:

PYTHON

```
def require_attempt(record: dict[str, object]) -> Attempt:
    team = record.get("team")
    clues_found = record.get("clues_found")
    elapsed_minutes = record.get("elapsed_minutes")
    bonus_clue = record.get("bonus_clue")

    if not isinstance(team, str) or not team.strip():
        raise ValueError("team must be non-empty text")
    if isinstance(clues_found, bool) or not isinstance(clues_found, int):
        raise ValueError("clues_found must be an integer")
    if isinstance(elapsed_minutes, bool) or not isinstance(elapsed_minutes, int):
        raise ValueError("elapsed_minutes must be an integer")
    if not isinstance(bonus_clue, bool):
        raise ValueError("bonus_clue must be boolean")

    return Attempt(
        team=team,
        clues_found=clues_found,
        elapsed_minutes=elapsed_minutes,
        bonus_clue=bonus_clue,
    )
```

Notice the explicit boolean checks for integer fields. At runtime, `bool` is a subclass of `int`, so `isinstance(True, int)` is true. The domain rejects that otherwise surprising value.

Annotations document the verified result of the boundary. Runtime checks make that claim true. Tests exercise representative behavior. The three forms of evidence reinforce rather than replace one another.

Checkpoint: structured data and validation

Structured records at a real boundary

Choose the type or validation step supported by the code’s actual runtime shape.

1. What is an `Attempt` TypedDict value at runtime?

- **A.** A dataclass instance
- **B.** A tuple
- **C.** A new dictionary subclass with automatic validation
- **D.** An ordinary dictionary whose expected keys and values can be checked statically

2. A JSON object arrives as `dict[str, object]`. What should happen before returning it as `Attempt`?

- **A.** Validate and narrow each required field, then construct the typed record
- **B.** Use a broad ignore so MyPy accepts it
- **C.** Change every value annotation to Any

- **D.** Assume valid JSON has the correct domain types

3. When is a frozen dataclass usually clearer than TypedDict?

- **A.** When data must remain JSON text
- **B.** When the program wants a domain object with named attributes, constructor behavior, and value equality
- **C.** When keys are unknown until runtime
- **D.** When no fields have types

Write your choices before checking the answer key at the end of this book.

7. Lab: give the museum package an honest contract

Annotate this small package boundary without changing its behavior:

```python
def rank_attempts(attempts):
    results = []
    for attempt in attempts:
        results.append(
            {
                "team": attempt["team"],
                "score": score_attempt(attempt),
                "elapsed_minutes": attempt["elapsed_minutes"],
            }
        )
    return sorted(
        results,
        key=lambda result: (
            -result["score"],
            result["elapsed_minutes"],
            result["team"],
        ),
    )
```

Requirements:

- define `Attempt` with the four fields used earlier;
- define a `Result` record containing `team`, `score`, and `elapsed_minutes`;
- accept any reusable, ordered or unordered sequence that the function only iterates; choose and justify the abstract input type;
- return `list[Result]`;
- preserve alphabetical ascending team names after equal score and elapsed time;
- add an `optional_winner(results)` function returning `Result | None`;
- keep runtime validation outside `rank_attempts`; it receives verified attempts; and
- run the behavior assertions after annotating.

```python
attempts = [
    {"team": "Moon Moths", "clues_found": 6, "elapsed_minutes": 42, "bonus_clue": True},
    {"team": "Brass Bats", "clues_found": 7, "elapsed_minutes": 50, "bonus_clue": False},
]
ranked = rank_attempts(attempts)
```

PYTHON (CONTINUED)

```
assert ranked[0]["team"] == "Moon Moths"
assert optional_winner(ranked) == ranked[0]
assert optional_winner([]) is None
```

Hint A: choose the input interface

The function only iterates over attempts. `Iterable[Attempt]` is sufficient and allows a list, tuple, or generator. If you intend to promise reusable ordered input to callers, `Sequence[Attempt]` is also honest but stronger than necessary for this implementation.

Hint B: construct a TypedDict result

After defining `class Result(TypedDict)`, use `Result(team=..., score=..., elapsed_minutes=...)`. It creates an ordinary dictionary while making the intended keys visible to the checker.

Hint C: handle the empty result explicitly

Check `if not results: return None` before returning `results[0]`. The check narrows the behavior and prevents `IndexError`.

Show one complete annotated solution

PYTHON

```
from collections.abc import Iterable, Sequence
from typing import TypedDict


class Attempt(TypedDict):
    team: str
    clues_found: int
    elapsed_minutes: int
    bonus_clue: bool


class Result(TypedDict):
    team: str
    score: int
    elapsed_minutes: int


def rank_attempts(attempts: Iterable[Attempt]) -> list[Result]:
    results: list[Result] = []
    for attempt in attempts:
        results.append(
            Result(
                team=attempt["team"],
                score=score_attempt(attempt),
                elapsed_minutes=attempt["elapsed_minutes"],
            )
        )
    return sorted(
        results,
        key=lambda result: (
            -result["score"],
            result["elapsed_minutes"],
            result["team"],
        ),
    )


def optional_winner(results: Sequence[Result]) -> Result | None:
    if not results:
        return None
    return results[0]
```

`Iterable` is enough for ranking because the function traverses once. `Sequence` is useful for `optional_winner` because it checks length/truth and indexes position zero.

Key points

- Annotations communicate static expectations; Python does not automatically validate or convert arguments at runtime.
- Annotate public boundaries and ambiguous empty values; let a checker infer obvious locals.
- Use `T | None` when absence is real, then handle it explicitly.
- Choose `Iterable`, `Collection`, `Sequence`, `Mapping`, or a concrete mutable type from the operations the function needs.
- TypedDict describes dictionary-shaped data; a dataclass creates a runtime domain object.
- Validate untrusted values before claiming a narrow type. Types, validation, and tests answer different questions.

References

- Python `typing` module
- Typing best practices
- Python typing specification: Typed dictionaries
- Python collections abstract base classes

Continue to Lesson 3: Read MyPy Errors and Close Type Gaps

Read MyPy Errors and Close Type Gaps

Online lesson and runnable notebook

Run MyPy in strict mode, trace diagnostics through inference and narrowing, and repair real contract mismatches without broad ignores.

- **Level:** Python Foundations
- **Estimated time:** 3–4 hours
- **You will learn:** Follow MyPy's evidence from a declared contract to the incompatible value, narrow uncertain types, and keep unchecked `Any` from spreading.
- **Practice in:** The Midnight Museum local project with MyPy 2.2-compatible examples

1. A diagnostic is a trail, not a verdict

Suppose the museum package declares this function:

PYTHON
```
def overtime_penalty(elapsed_minutes: int) -> int:
    return max(0, elapsed_minutes - 45) * 2
```

A caller reads a text field but passes it without conversion:

PYTHON
```
raw_minutes: str = "50"
penalty = overtime_penalty(raw_minutes)
```

Both snippets are valid Python syntax. Running the caller would eventually raise `TypeError` in subtraction. MyPy can compare the declared types without executing the call:

BASH
```
mypy src
```

TEXT
```
src/midnight_museum/report.py:4: error: Argument 1 to "overtime_penalty" has incompatible type "str"; e ↪
    xpected "int"  [arg-type]
Found 1 error in 1 file (checked 3 source files)
```

Before changing anything, answer these questions:

1. Which file and line contains the incompatible call?
2. What type did MyPy infer or read for the supplied value?
3. What type does the function contract require?
4. Is the annotation wrong, or did validation/conversion fail at the caller?
5. Which focused command will prove that the repair closes this diagnostic?

Read the message from left to right:

Part	Evidence
`src/midnight_museum/report.py`	File containing the reported use
`:4`	Line where MyPy can demonstrate the mismatch
`error`	Severity
`Argument 1 ...`	Operation being checked
`str`; expected `int`	Actual and required static types
`[arg-type]`	Stable error-code family for documentation or a precise ignore

The diagnostic line is not always the original source of the bad assumption. Here the boundary accepted text earlier. Repairing `overtime_penalty` to accept `str` would weaken a clear numeric contract. Validate or parse before the call:

```python
def parse_elapsed_minutes(raw_minutes: str) -> int:
    minutes = int(raw_minutes)
    if minutes < 0:
        raise ValueError("elapsed minutes cannot be negative")
    return minutes


raw_minutes = "50"
penalty = overtime_penalty(parse_elapsed_minutes(raw_minutes))
assert penalty == 10
```

Then rerun the smallest useful target and the project target:

```bash
mypy src/midnight_museum/report.py
mypy src
```

2. Let inference do useful work, then inspect it deliberately

MyPy follows assignments and return paths even when every local has no explicit annotation:

```python
def score_label(score: int) -> str:
    points_after_bonus = score + 25
    return f"{points_after_bonus} points"
```

It infers `points_after_bonus` as `int` from `score + 25`. If you are unsure what it knows, use `reveal_type` temporarily inside a static-only branch:

```python
from typing import TYPE_CHECKING


def score_label(score: int) -> str:
    points_after_bonus = score + 25
    if TYPE_CHECKING:
        reveal_type(points_after_bonus)
    return f"{points_after_bonus} points"
```

```text
src/midnight_museum/scoring.py:7: note: Revealed type is "builtins.int"
Success: no issues found in 1 source file
```

`TYPE_CHECKING` is false at ordinary runtime, so the undefined MyPy helper is not called. Remove the investigation after it answers the question; a project full of stale `reveal_type` notes becomes noise.

Inference is local evidence, not a reason to omit public contracts. Without parameter annotations, a function can become dynamically typed:

```python
def unsafe_label(score):
    return score.missing_method() + 1
```

In permissive settings, `score` may become `Any`, and almost every operation is accepted. Strict mode reports the missing annotations and checks more of the body.

Checkpoint: read the diagnostic

Follow MyPy's evidence

Use the shown diagnostic fields and contracts to choose the repair.

1. The message says an argument has type `str` but `overtime_penalty` expects `int`. What is the strongest first investigation?

- **A.** Delete the return annotation
- **B.** Ignore every arg-type error
- **C.** Find where the text enters and parse or validate it before the numeric boundary
- **D.** Change the penalty formula

2. What does `[arg-type]` identify in the diagnostic?

- **A.** The Python version
- **B.** The runtime exception class
- **C.** The test node ID
- **D.** The MyPy error-code family for an incompatible argument

3. When is temporary `reveal_type` useful?

- **A.** When you need to inspect the type MyPy inferred at a specific expression
- **B.** To convert a value at runtime
- **C.** To install missing dependencies
- **D.** To replace a public annotation

Write your choices before checking the answer key at the end of this book.

3. Narrow a union along the same branches the runtime checks

MyPy rejects a string operation while `winner` might still be `None`:

```python
def winner_name(scores: dict[str, int]) -> str | None:
    if not scores:
        return None
    return max(scores, key=scores.get)


winner = winner_name({})
```

Trying `winner.upper()` would produce a union diagnostic because `None` has no `upper`. Narrow it with the real runtime decision:

```python
winner = winner_name({"Moon Moths": 205})
if winner is None:
    heading = "NO WINNER"
else:
    heading = winner.upper()

assert heading == "MOON MOTHS"
```

Inside `else`, the impossible `None` case has been removed. The same idea works with `object` and `isinstance`:

```python
def require_score(value: object) -> int:
    if isinstance(value, bool):
        raise ValueError("score cannot be boolean")
    if not isinstance(value, int):
        raise ValueError("score must be an integer")
    return value


assert require_score(205) == 205
```

The second branch narrows `value` to `int`. The boolean check comes first because booleans are integer subclasses at runtime.

Exhaust every declared case

A small literal vocabulary can be checked with an explicit final assertion:

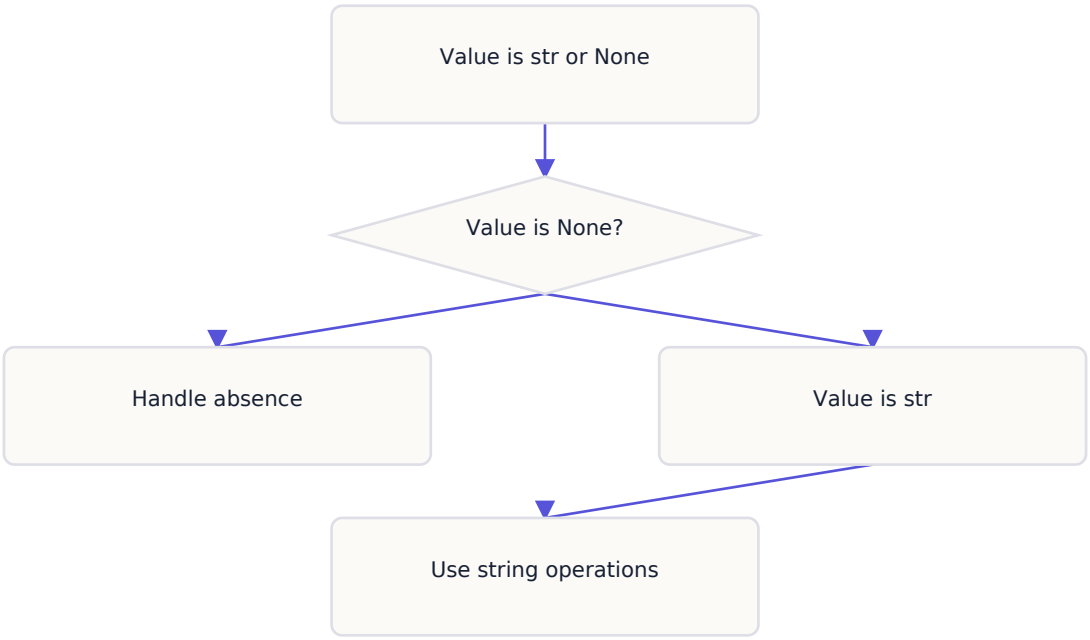
```python
from typing import Literal

Decision = Literal["winner", "runner-up", "participant"]


def badge_for(decision: Decision) -> str:
    if decision == "winner":
        return "gold"
    if decision == "runner-up":
        return "silver"
    if decision == "participant":
        return "blue"
    raise AssertionError("unreachable decision")
```

For this foundations lesson, the explicit branches are enough. Later Python and typing tools offer specialized exhaustive-check helpers, but do not introduce a new abstraction when the small vocabulary is already clear.

Each runtime branch removes impossible members from the static union.



Value is None? → Handle absence: Yes

Value is None? → Value is str: No

4. Treat Any as a gap in the fence

`Any` is compatible with every type in both directions. That makes it useful at rare integration boundaries and dangerous as a default:

```python
from typing import Any


def unchecked_score(payload: Any) -> int:
    return payload["result"].mystery().points
```

A checker permits the indexing, nonexistent method, attribute access, and `int` return claim because each value derived from `Any` is usually also `Any`. The code can fail anywhere at runtime.

`object` is safer when the boundary truly accepts any object:

```python
def checked_score(payload: object) -> int:
    if not isinstance(payload, dict):
        raise ValueError("payload must be a dictionary")

    raw_score = payload.get("score")
    if isinstance(raw_score, bool) or not isinstance(raw_score, int):
        raise ValueError("score must be an integer")
    return raw_score
```

With `object`, MyPy allows only operations valid for all objects until runtime checks narrow it. Use `Any` when interoperability genuinely requires opting out of static checking, not because a precise type takes another minute to design.

Find how Any entered

Common paths include:

- an unannotated function parameter or return value;
- a third-party library without type information;
- JSON decoded and immediately claimed to be a domain record;
- an overbroad `cast` or ignore; and
- a library configured with `ignore_missing_imports` for every module.

Strict settings such as `disallow_untyped_defs` and `warn_return_any` help stop unchecked values from crossing public boundaries. Do not respond by annotating everything as `Any`; that satisfies syntax while removing the evidence you wanted.

Checkpoint: narrowing and Any

Close uncertain paths deliberately

Choose the branch or type that preserves the strongest useful checking.

1. After `if winner is None: ... else:`, what type can MyPy use for `winner` inside the else branch?

- **A.** object
- **B.** str
- **C.** None
- **D.** Any

2. Why is `object` safer than `Any` for an untrusted boundary?

- **A.** object converts every input
- **B.** object is always a dictionary
- **C.** A checker requires narrowing before operations that are not valid for every object
- **D.** object disables runtime validation

3. What usually happens to a value returned by a method called on an `Any` value?

- **A.** It becomes None
- **B.** It is inferred as str
- **C.** MyPy rejects every method
- **D.** It is usually also Any, allowing the unchecked gap to propagate

Write your choices before checking the answer key at the end of this book.

5. Put strict policy in the project

Command-line flags are easy to forget. Commit the shared contract in `pyproject.toml`:

```toml
[tool.mypy]
python_version = "3.10"
```

TOML (CONTINUED)

```
strict = true
show_error_codes = true
warn_unused_ignores = true
warn_redundant_casts = true
warn_unused_configs = true
```

`strict = true` enables a documented bundle of stricter checks. The bundle may evolve across major MyPy versions, which is one reason the development dependency and lock matter. The other warnings catch configuration or escape hatches that no longer serve a purpose.

Run the same target from the project root:

BASH

```
mypy src
```

TEXT

```
Success: no issues found in 2 source files
```

A clean result means MyPy found no issue covered by the configured checks. It does not mean inputs have been validated, branches have been tested, or the business rule is correct.

Adopt checking in manageable slices

For an older untyped project, turning on strict mode for everything can produce hundreds of interacting messages. Start with a coherent boundary:

1. annotate a widely used public function;
2. annotate the data entering and leaving it;
3. remove the `Any` gaps revealed nearby;
4. run a focused MyPy target;
5. add that target to the quality gate; and
6. expand without allowing new unchecked functions in the completed area.

A narrow configuration override can document a temporary boundary:

TOML

```
[[tool.mypy.overrides]]
module = ["legacy_importer"]
ignore_errors = true
```

This is visible debt, not a finished solution. Prefer a module-scoped migration over global `ignore_missing_imports = true`, which can hide misspelled imports and missing type information everywhere.

6. Understand library types and precise ignores

A typed installed package may include inline annotations and a `py.typed` marker, or supply separate stub files ending in `.pyi`. A checker uses those records to understand imported functions. If no information is available, MyPy may report an import or treat values as `Any`, depending on configuration.

Do not silence every import problem first. Check:

1. Is the import name spelled correctly?
2. Is the dependency installed in the same environment as MyPy?
3. Does the library ship types or recommend a stub package?
4. Is a small local boundary wrapper easier to validate than allowing `Any` through the project?

If an ignore is truly required, name the exact code and explain the external constraint:

```python
from old_museum_reader import load_attempts  # type: ignore[import-untyped]
```

A precise ignore can become stale. With `warn_unused_ignores`, MyPy reports it when the library later gains type information.

Avoid changing an expected return annotation merely to make a real implementation error disappear:

```python
def team_label(team_name: str) -> str:
    return 42
```

The right repair is to return text, not claim the public function returns `str | int` unless both types are genuine behavior:

```python
def team_label(team_name: str) -> str:
    return f"Team: {team_name}"


assert team_label("Moon Moths") == "Team: Moon Moths"
```

Checkpoint: project policy and repair choices

Keep the checker trustworthy

Choose the action that preserves an honest project contract.

1. `team_label` is declared to return str but returns 42. Which repair best matches its stated purpose?

- **A.** Return formatted text and keep `-> str`
- **B.** Change the return to Any
- **C.** Add a global ignore
- **D.** Delete the function’s tests

2. If an untyped legacy dependency genuinely requires an ignore, which form is most reviewable?

- **A.** # type: ignore on every line
- **B.** A narrow `# type: ignore[import-untyped]` with the external reason
- **C.** strict = false for the whole project
- **D.** Annotate every returned value as Any

3. What does `Success: no issues found` demonstrate?

- **A.** All runtime inputs are valid
- **B.** Every branch has tests
- **C.** The configured MyPy checks found no type issue in the selected target
- **D.** The program has no defects

Write your choices before checking the answer key at the end of this book.

7. Lab: close seven type gaps

Place this valid Python in `src/midnight_museum/repairs.py`, enable the strict configuration above, and run `mypy src/midnight_museum/repairs.py`:

PYTHON

```
from typing import Any


def first_team(teams):
    if teams:
        return teams[0]


def score_from_payload(payload: Any) -> int:
    return payload["score"]


def render_score(score: int) -> str:
    return score


def add_bonus(score: int, bonus: str) -> int:
    return score + bonus


winner = first_team(["Moon Moths"])
print(winner.upper())
```

Repair these seven gaps without weakening the intended behavior:

1. annotate `teams` with the smallest interface needed for truth and position zero;
2. make `first_team` declare and return the absent case explicitly;
3. replace the `Any` payload with `object` and validate a dictionary integer field;
4. return text from `render_score`;
5. give `bonus` its actual numeric type;
6. narrow `winner` before calling `.upper()`; and
7. keep strict mode and finish with both a clean MyPy run and behavior assertions.

Hint A: start at the outer signatures

Use `Sequence[str]` for teams and `str | None` for the search result. Use `object` for the untrusted payload so every operation must be justified.

Hint B: narrow the payload in stages

First require `dict`; then use `.get("score")`; reject booleans and require an integer before returning the value.

Hint C: do not hide the winner case

Use an explicit `if winner is None` branch. The ordinary branch can safely call `.upper()` after the check.

Show a complete strict solution

PYTHON
```
from collections.abc import Sequence


def first_team(teams: Sequence[str]) -> str | None:
    if not teams:
        return None
    return teams[0]


def score_from_payload(payload: object) -> int:
    if not isinstance(payload, dict):
        raise ValueError("payload must be a dictionary")
    score = payload.get("score")
    if isinstance(score, bool) or not isinstance(score, int):
        raise ValueError("score must be an integer")
    return score


def render_score(score: int) -> str:
    return f"{score} points"


def add_bonus(score: int, bonus: int) -> int:
    return score + bonus


winner = first_team(["Moon Moths"])
if winner is None:
    winner_heading = "NO WINNER"
else:
    winner_heading = winner.upper()

assert winner_heading == "MOON MOTHS"
assert score_from_payload({"score": 205}) == 205
assert render_score(205) == "205 points"
assert add_bonus(180, 25) == 205
```

Run:

BASH
```
mypy src/midnight_museum/repairs.py
pytest -q
```

Do not accept the repair until static contracts and checked behavior both pass.

Key points

- Read a MyPy diagnostic as a trail from reported use to actual and expected types; the highlighted line may not be the original boundary mistake.
- Use inference for obvious locals and temporary `reveal_type` to inspect a confusing path.
- Narrow unions and `object` with the same runtime branches that make operations safe.
- `Any` disables useful checking and tends to propagate; locate and contain its entry point.

- Commit strict project policy, run a consistent target, and prefer precise, explained ignores over global silence.
- A clean MyPy run complements runtime validation and tests; it replaces neither.

References

- MyPy: Getting started
- MyPy: Type narrowing
- MyPy: Dynamically typed code and Any
- MyPy configuration file
- Typing specification: Distributing type information

Continue to Lesson 4: Format and Lint a Project with Ruff

Format and Lint a Project with Ruff

Online lesson and runnable notebook

Configure Ruff, interpret rule diagnostics, review automatic fixes, and separate deterministic layout from human design and behavior evidence.

- **Level:** Python Foundations
- **Estimated time:** 3–4 hours
- **You will learn:** Run Ruff's formatter and linter as distinct tools, trace findings to configured rules, and turn a noisy file into a reviewed patch.
- **Practice in:** The Midnight Museum local project, terminal, and Git diff

1. Decide which question you are asking

A teammate sends this museum helper:

```python
import os
import json
from pathlib import Path


def load_team(path: Path):
    data = json.loads(path.read_text())
    return data
```

Before running a tool, ask:

1. Is the layout consistent with the project?
2. Does static source contain a suspicious or unused construct?
3. Do the declared types agree across calls?
4. Does the function return the correct data for real files and failures?
5. Is the interface understandable to a future maintainer?

Those are different questions:

- `ruff format` rewrites deterministic layout.
- `ruff check` reports configured lint rules and can fix selected findings.
- MyPy compares static type contracts.
- pytest executes behavioral examples.
- a human decides whether `load_team` has a useful name, contract, error policy, and responsibility.

Do not treat a clean result from one as a substitute for the others.

From the project root, run the linter without changing files:

```bash
ruff check src tests
```

For the helper above, Ruff 0.15.20 with the course project's rules reports:

```text
I001 [*] Import block is un-sorted or un-formatted
 --> src/midnight_museum/storage.py:1:1
help: Organize imports

F401 [*] `os` imported but unused
 --> src/midnight_museum/storage.py:1:8
help: Remove unused import: `os`

Found 2 errors.
[*] 2 fixable with the `--fix` option.
```

The file is valid Python and might even pass a happy-path test. The report gives a different form of evidence.

2. Read a Ruff finding before fixing it

Each diagnostic tells you:

Field	Meaning in the sample
`I001`	Rule code; `I` is the import-sorting family
`[*]`	Ruff marks an automatic fix as available
Message	The import block is not in configured order
Path and `1:1`	File, line, and column where the finding is anchored
Help	Proposed action, not proof that the action fits every intention

`F401` comes from the Pyflakes-compatible `F` family. It says the imported name is unused in this module. Usually deleting `os` is correct. Sometimes an import is deliberately re-exported from a package. In that case, make the public interface explicit—often through `__all__`—rather than accumulating unexplained ignores.

Run one rule or one file while investigating

Focus shortens the feedback loop:

```bash
ruff check src/midnight_museum/storage.py
ruff check src/midnight_museum/storage.py --select F401
ruff rule F401
```

The first selects a file, the second selects a rule for this run, and the third opens Ruff's local explanation. Command-line selection is useful for diagnosis; the committed project configuration remains the shared policy.

Common rule families in this course

The repository selects a deliberate, moderate set:

Prefix	Kind of evidence	Example
`E`	pycodestyle errors	ambiguous whitespace or a configured long line
`F`	Pyflakes correctness signals	undefined or unused names
`I`	import organization	standard-library and local imports out of order
`UP`	syntax compatible with the target Python	older typing syntax that can be modernized
`B`	bug-prone patterns	a mutable default or loop-variable capture
`RUF`	Ruff-specific checks	ambiguous or unsafe source patterns

A rule family is not inherently right for every project. Select it because the team understands and wants its policy.

Checkpoint: classify Ruff evidence

Read the Ruff report

Choose the conclusion supported by the shown command and diagnostic.

1. What does `F401 os imported but unused` demonstrate?

- **A.** The program failed every runtime test
- **B.** The os package is broken
- **C.** The file cannot be parsed
- **D.** The configured linter found an imported name with no use in that module

2. Which command investigates only F401 in one file without editing it?

- **A.** ruff check src/midnight_museum/storage.py –select F401
- **B.** ruff format –check
- **C.** pytest -q
- **D.** mypy –strict

3. Which question remains primarily a human design decision after Ruff is clean?

- **A.** Whether imports follow configured ordering
- **B.** Whether `load_team` owns an understandable responsibility and error policy
- **C.** Whether the file has unused imports
- **D.** Whether configured layout is stable

Write your choices before checking the answer key at the end of this book.

3. Let the formatter own mechanical layout

This function is valid Python but inconsistent with the project:

PYTHON
```
def museum_label( team:str,score:int )->str:
    return f"{team}:{score}"
```

Run the formatter in place:

BASH
```
ruff format src/midnight_museum/labels.py
```

The result is deterministic for the selected Ruff version and configuration:

PYTHON
```
def museum_label(team: str, score: int) -> str:
    return f"{team}:{score}"
```

To check without rewriting—appropriate for a gate—use:

BASH
```
ruff format --check src tests
```

A file needing changes produces a nonzero exit status. A clean run looks like:

TEXT
```
3 files already formatted
```

Review formatting as a diff

Even a deterministic tool can touch many lines. Use Git to see the patch:

BASH
```
git diff -- src tests
```

Check that only layout changed, then run behavior tests. A formatter aims to preserve behavior, but project confidence comes from the complete evidence chain, not trust in a slogan.

Do not fight the formatter line by line

Ruff’s formatter deliberately has a small configuration surface. Agree on a line length and a few stable preferences rather than hand-arranging every expression. If a formatted expression is difficult to read, improve the source structure—extract an intermediate value or function—instead of adding fragile spacing.

PYTHON
```
def score_attempt(attempt):
    clue_points = attempt["clues_found"] * 30
    bonus_points = 25 if attempt["bonus_clue"] else 0
    overtime_penalty = max(0, attempt["elapsed_minutes"] - 45) * 2
    return max(0, clue_points + bonus_points - overtime_penalty)
```

The names solve a reading problem that line wrapping cannot.

4. Apply fixes in an order you can explain

Ruff can apply marked safe fixes:

```bash
ruff check src tests --fix
ruff format src tests
```

For the import sample, the first command removes `os` and sorts the remaining imports:

```python
import json
from pathlib import Path


def load_team(path: Path):
    data = json.loads(path.read_text())
    return data
```

Run lint fixes before formatting because a fix may reorganize imports or source that then needs formatting. After both commands:

```bash
git diff --check
ruff check src tests
ruff format --check src tests
pytest -q
```

Safe does not mean semantically proven

A “safe fix” means Ruff classifies it as preserving runtime behavior for the rule’s model. It cannot know external reflection, unusual import side effects, or every dynamic dependency. Review the diff.

Ruff can expose fixes it labels unsafe, but do not enable them as a blanket shortcut:

```bash
ruff check src tests --unsafe-fixes
```

This displays availability; adding `--fix --unsafe-fixes` would apply them. Use an unsafe fix only after reading the rule, understanding the behavior risk, and having checks that cover the affected contract.

Fix the cause or document a narrow exception

Suppose a pytest fixture intentionally receives an unused conventional argument. A per-file or rule-specific exception can be clearer than renaming values throughout production code. But an unexplained global ignore trains future maintainers to disregard useful output.

Use this decision order:

1. Is the finding a real bug or unnecessary construct? Fix the source.
2. Does a safe mechanical fix match the intention? Apply and review it.
3. Is the rule wrong for the whole project? Change the explicit policy with a reason and review.
4. Is one file genuinely different? Add the narrowest documented exception.
5. Never change configuration merely to make a dashboard green.

Checkpoint: formatter and fixer workflow

Turn findings into a reviewed patch

Choose the command or sequence that preserves the clearest evidence.

1. Which command verifies formatting without rewriting source?

- **A.** ruff check –fix
- **B.** git diff
- **C.** ruff format –check src tests
- **D.** pytest -q

2. Why should Ruff lint fixes usually run before Ruff formatting?

- **A.** Formatting disables linting
- **B.** Tests cannot run after formatting
- **C.** MyPy requires unsorted imports
- **D.** Lint fixes may rewrite imports or source that then needs deterministic formatting

3. What is the responsible response to an available unsafe fix?

- **A.** Read the rule and behavior risk, apply only deliberately, review the diff, and rerun checks
- **B.** Enable all unsafe fixes permanently
- **C.** Assume unsafe means syntax error
- **D.** Disable every rule in the family

Write your choices before checking the answer key at the end of this book.

5. Commit the policy in pyproject.toml

A shared project should not depend on one person’s editor settings. Put the policy beside other project metadata:

```toml
[tool.ruff]
line-length = 88
target-version = "py310"
src = ["src", "tests"]

[tool.ruff.lint]
select = [
  "E",
  "F",
  "I",
  "UP",
  "B",
  "RUF",
]

[tool.ruff.format]
quote-style = "double"
```

Understand each field

- `line-length` guides wrapping and relevant lint rules; it is not permission to compress every statement up to character 88.
- `target-version` lets Ruff avoid syntax unavailable to the oldest supported Python. It must agree with `[project].requires-python`.
- `src` helps import classification for a `src` layout.
- `select` makes the chosen rule families explicit. Ruff documentation advises adding families deliberately; `ALL` silently grows when a new release adds rules.
- formatter options choose a few stable preferences without attempting to encode personal layout for every construct.

Ruff discovers `pyproject.toml`, `ruff.toml`, or `.ruff.toml` by walking from a file toward parent directories. Run from the expected project root and inspect configuration when a result surprises you:

```BASH
ruff check src/midnight_museum/quest.py --show-settings
```

File discovery and Git ignores

By default, current Ruff versions discover Python files and Jupyter notebooks and respect common ignore files. Passing a file explicitly can override ordinary exclusion unless `force-exclude` is configured. This distinction explains why “Ruff skipped the generated directory during a project run” and “Ruff checked a named generated file” can both be true.

Use tool-specific exclusion when lint and format policy differ:

```TOML
[tool.ruff.format]
exclude = ["generated/*.py"]
```

Avoid excluding tests merely because they reveal findings. Tests can have narrow, justified differences:

```TOML
[tool.ruff.lint.per-file-ignores]
"tests/**" = ["S101"]
```

Only add a rule that is actually selected and understood. `S101` is shown as a configuration-shape example; it is not selected by this unit’s base policy.

6. Diagnose configuration instead of guessing

Three common surprises have different causes:

“The formatter passed, but lint failed”

Expected. Formatting handles layout; lint rules inspect other source patterns. Run and interpret both.

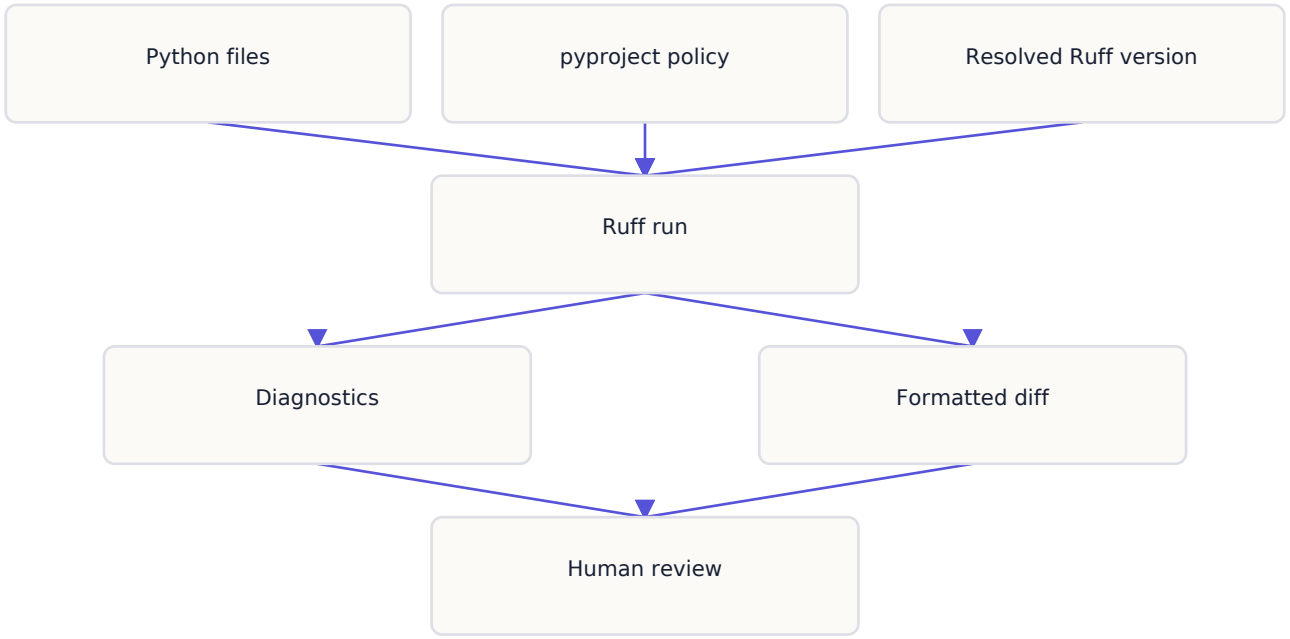
“My editor and terminal disagree”

Check whether the editor uses the project configuration and the same Ruff version. An editor-specific override can take precedence. Run the committed command in the declared environment as the reproducible result.

“A rule appeared after an upgrade”

An explicit rule family can gain rules in a later tool release. Review the release and new diagnostic, then either repair, configure a reasoned exception, or pin/update the tool deliberately. Do not combine a large tool upgrade with an unrelated refactor if you want a reviewable diff.

Committed configuration and a resolved tool version produce the repeatable Ruff result.



Checkpoint: configuration boundaries

Make Ruff reproducible

Choose the configuration decision that best explains or controls the project result.

1. Why should Ruff’s target-version agree with the project’s minimum Python version?

- **A.** It changes pytest discovery
- **B.** It prevents Ruff from recommending or formatting syntax unavailable to supported interpreters
- **C.** It installs Python automatically
- **D.** It makes all dependencies typed

2. Why prefer an explicit small `select` list over enabling `ALL` without review?

- **A.** ALL disables fixes
- **B.** ALL checks only Markdown
- **C.** New releases can add rules, silently changing the project’s policy
- **D.** Explicit lists make Ruff slower

3. The editor passes but the terminal gate fails. What should you compare first?

- **A.** Team names in test data

- **B.** Only the operating-system theme
- **C.** The number of comments
- **D.** The Ruff version, discovered project configuration, target path, and editor overrides

Write your choices before checking the answer key at the end of this book.

7. Lab: turn six findings into a reviewed patch

Create `src/midnight_museum/cleanup.py` with this valid source:

```python
import os
import json
from pathlib import Path


BONUS = 25


def load_score(path: Path):
    data = json.loads(path.read_text())
    score = data["clues"] * 30
    if data["bonus"] == True:
        score += BONUS
    return score


def labels(names=[]):
    names.append("MIDNIGHT MUSEUM")
    return names
```

Use the unit configuration, then:

1. run `ruff check` without fixes and classify every finding by rule code;
2. predict which findings `--fix` will change;
3. apply safe fixes, then run `ruff format`;
4. inspect the diff before editing remaining findings;
5. replace the mutable default with `None` and create a fresh list;
6. replace the explicit `== True` comparison with the boolean condition;
7. add useful parameter/return annotations without turning this into the MyPy lesson again;
8. run lint, format check, MyPy, and pytest; and
9. explain one choice Ruff made and one design choice you made.

Hint A: start with imports

`os` is unused and the import block is out of order. Ruff can remove and sort those safely. Review the resulting import group.

Hint B: repair the mutable default

Use `def labels(names: list[str] | None = None) -> list[str]:`, then create `result = [] if names is None else list(names)`. Returning a copy also avoids mutating a caller's supplied list.

Hint C: finish with check-only commands

After edits, run `ruff check src tests`, `ruff format --check src tests`, `mypy src`, and `pytest -q`. The final gate should not rewrite anything.

Show one reviewed result

```python
import json
from pathlib import Path

BONUS_POINTS = 25


def load_score(path: Path) -> int:
    data = json.loads(path.read_text())
    score = data["clues"] * 30
    if data["bonus"]:
        score += BONUS_POINTS
    return score


def labels(names: list[str] | None = None) -> list[str]:
    result = [] if names is None else list(names)
    result.append("MIDNIGHT MUSEUM")
    return result
```

The automatic pass can remove/sort imports and normalize layout. A maintainer chooses the name `BONUS_POINTS`, the mutable-default repair, copy behavior, and boundary annotations. Add runtime JSON validation in the appropriate boundary lesson/project rather than claiming the decoded object is statically safe.

Key points

- Formatter, linter, type checker, tests, and human review answer different questions.
- Read path, position, rule code, message, and fix status before changing code.
- Apply lint fixes before formatting, inspect the diff, and rerun check-only commands.
- Safe fixes are still reviewed changes; unsafe fixes require explicit behavior reasoning.
- Commit a small understood policy and a compatible tool version instead of relying on editor state.
- A narrow justified exception is better than a global ignore, but repairing the source is usually the first choice.

References

- Ruff linter
- Ruff formatter
- Configuring Ruff
- Ruff rules reference
- Ruff integrations and tool ordering

Continue to Lesson 5: Catch Small Problems Before Commit

Catch Small Problems Before Commit

Online lesson and runnable notebook

Configure and diagnose pre-commit hooks so staged changes receive fast, visible feedback before they reach shared review.

- **Level:** Python Foundations
- **Estimated time:** 3–4 hours
- **You will learn:** Install, run, and debug a local pre-commit gate, including staged-file selection, tool modifications, failed hooks, and reproducible configuration.
- **Practice in:** A local Git checkout of the Midnight Museum project

1. Move feedback closer to the edit

By the end of Lesson 4, the museum project has a reliable manual sequence:

```bash
ruff check src tests
ruff format --check src tests
mypy src
pytest -q
```

The sequence works only when someone remembers to run it. A Git pre-commit hook runs after `git commit` begins but before Git creates the commit. The **pre-commit** framework manages those hook programs and their configuration.

Before installing anything, answer these questions:

1. Which files will an ordinary hook run inspect?
2. What happens if a formatter changes a file during the hook?
3. How does a hook communicate failure to Git?
4. Where are shared hook definitions stored, and where is the installed Git script stored?
5. Why does a local hook improve feedback without replacing CI?

Start with a Git repository and a committed configuration file named `.pre-commit-config.yaml`:

```yaml
repos:
  - repo: local
    hooks:
      - id: ruff-check
        name: Ruff lint
        entry: ruff check
        language: system
        types: [python]
      - id: ruff-format
        name: Ruff format
        entry: ruff format
        language: system
```

YAML (CONTINUED)

```
        types: [python]
      - id: mypy
        name: MyPy
        entry: mypy src
        language: system
        pass_filenames: false
      - id: pytest
        name: pytest
        entry: pytest -q
        language: system
        pass_filenames: false
        always_run: true
```

Install the Git hook script once per clone:

BASH

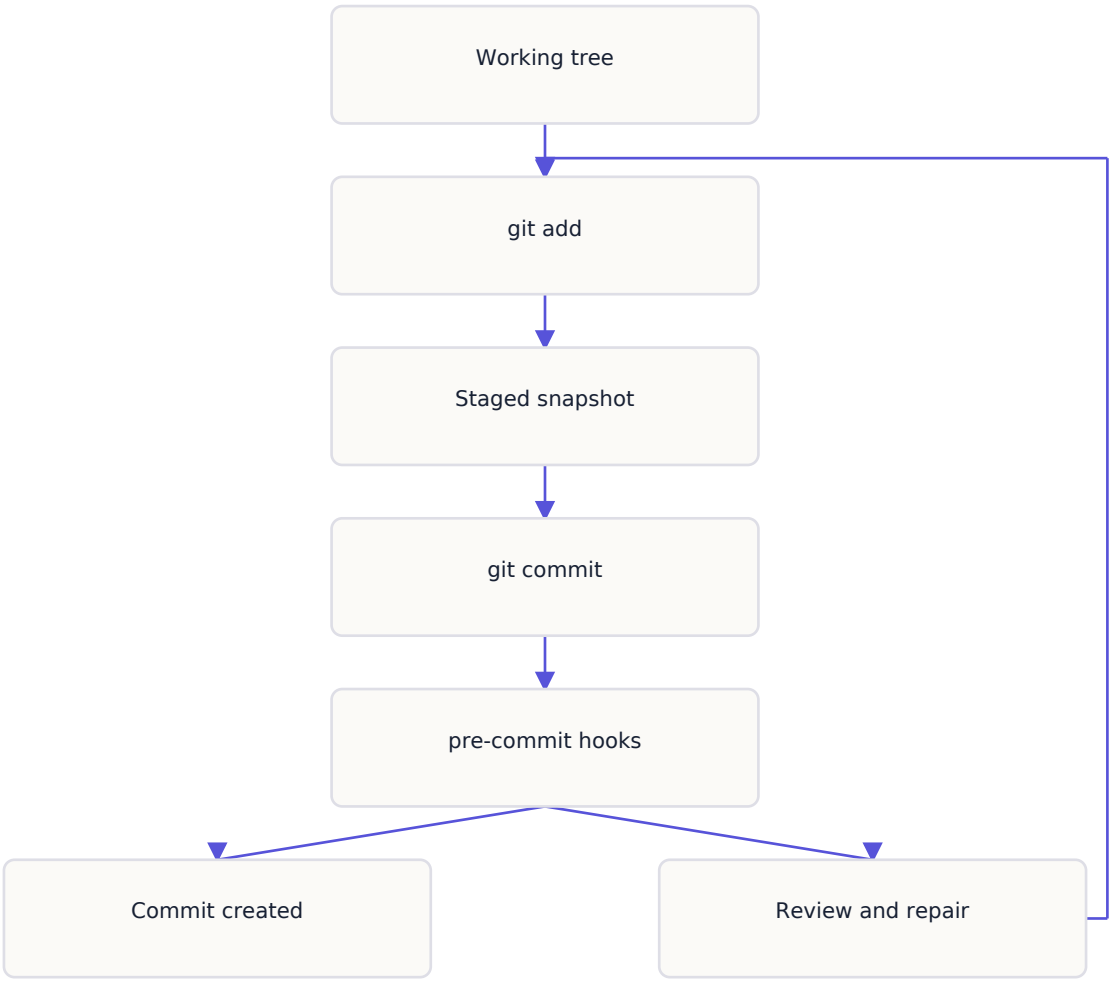
```
pre-commit install
```

TEXT

```
pre-commit installed at .git/hooks/pre-commit
```

The YAML belongs in version control. The installed `.git/hooks/pre-commit` script is local to this checkout. A new clone must run `pre-commit install` again.

The framework selects staged files, runs configured hooks, and lets Git continue only after a successful result.



pre-commit hooks → Commit created: Pass

pre-commit hooks → Review and repair: Fail or modify

2. Read configuration as an execution contract

Each field answers a practical question:

- `repos` lists sources of hook definitions.
- `repo: local` means this project supplies the command instead of downloading a hook repository.
- `id` is the stable name used to select one hook.
- `name` is the label shown in output.
- `entry` is the command executed; pre-commit does not split a shell pipeline for you unless the hook intentionally invokes a shell.
- `language: system` uses commands already available in the active environment.
- `types: [python]` filters filenames by identified file type.
- `pass_filenames: false` prevents pre-commit from appending selected filenames to a project-level command such as `mypy src` or `pytest -q`.

- `always_run: true` runs the test hook even when no selected Python filename would otherwise reach it.

Isolated hooks versus system commands

A remote hook repository can declare its own language environment and revision:

```yaml
repos:
  - repo: https://github.com/astral-sh/ruff-pre-commit
    rev: v0.15.20
    hooks:
      - id: ruff-check
        args: [--fix]
      - id: ruff-format
```

Pre-commit downloads and caches the pinned hook environment. This improves isolation and makes the revision visible. It also needs network access on the first run.

`language: system` is convenient when the project already installs all tools in one locked development environment. It assumes that environment is active and contains compatible commands. Neither style is universally correct; choose and document the environment boundary. The course repository uses system hooks for Ruff and MyPy because its Poetry/Conda development environment owns those tools.

Put rewrite-producing hooks first

If lint fixes or formatters change source, run them before check-only tools:

1. Ruff lint/fix;
2. Ruff formatting;
3. MyPy;
4. pytest.

A later check should examine the final source shape. In CI, use check-only commands because automation should report an unformatted commit rather than silently create an uncommitted correction.

Checkpoint: configuration anatomy

Read a hook configuration

Choose the field or trade-off supported by the YAML.

1. Why set `pass_filenames: false` for `entry: mypy src`?

- **A.** To prevent pre-commit from appending selected filenames to a command intentionally checking the whole src target
- **B.** To disable MyPy
- **C.** To make Git stage every file
- **D.** To run the command only on Markdown

2. What does `language: system` require?

- **A.** A Docker daemon

- **B.** The entry command must already exist in the active environment
- **C.** Every hook must be written in Bash
- **D.** The hook revision is downloaded automatically

3. Which file is shared, and which installation is local to a clone?

- **A.** Both files live only on GitHub
- **B.** The .git hook is committed and YAML is generated
- **C.** `.pre-commit-config.yaml` is committed; the script under `.git/hooks/` is installed locally
- **D.** Both are Python modules

Write your choices before checking the answer key at the end of this book.

3. Ordinary commits check the staged snapshot

Git distinguishes the working tree from the staged snapshot. If you edit `quest.py` and run `git add`, the current content becomes staged. If you edit it again without staging, the working tree and staged version differ.

Inspect both deliberately:

```bash
git status --short
git diff
git diff --cached
```

- `git diff` shows unstaged working-tree changes.
- `git diff --cached` shows the patch the next commit would contain.
- pre-commit temporarily manages unstaged changes so hooks can check the staged snapshot without mixing it with unrelated work.

A normal commit might show:

```text
Ruff lint.....................................................Passed
Ruff format...................................................Passed
MyPy..........................................................Passed
pytest........................................................Passed
```

A pass means each configured hook exited successfully for the selected state. It does not prove that every repository file was checked: file filters and the ordinary staged-file selection still matter.

Run the whole repository when adding or changing hooks

The first time a hook is introduced, existing unstaged files might contain the very problem it checks. Run:

```bash
pre-commit run --all-files
```

Run one hook while diagnosing:

```bash
pre-commit run ruff-check --all-files --verbose
```

`--verbose` displays more command output even when useful details would otherwise be hidden.

Passed, skipped, modified, and failed mean different things

```text
Ruff lint.....................................................Passed
Check YAML.....................................(no files to check)Skipped
Ruff format...................................................Failed
- hook id: ruff-format
- files were modified by this hook
MyPy..........................................................Failed
- hook id: mypy
- exit code: 1
```

- **Passed:** the hook returned success and made no disallowed change.
- **Skipped:** no selected file matched or the configured stage did not apply.
- **Modified/Failed:** a rewriting hook changed a file. Review the diff, stage the new content, and run again.
- **Failed with exit code:** the command found a problem or could not run. Read its diagnostic before editing.

Do not immediately rerun `git commit` after a formatter modification. First:

```bash
git diff
git add src/midnight_museum/quest.py
pre-commit run --all-files
```

The second clean run proves no hook has another change to make.

4. Diagnose the earliest surprising result

A hook says “command not found”

For a system hook, activate/install the declared development environment and confirm `command --version`. For an isolated remote hook, inspect environment creation output and network/cache availability.

A hook skips a file you expected

Check `files`, `exclude`, `types`, `types_or`, the Git stage, and whether the file is tracked. Use `pre-commit run <id> --files path/to/file.py --verbose` to test selection explicitly.

MyPy receives unexpected filenames

Project commands often need `pass_filenames: false`. Otherwise pre-commit appends selected files to the entry. Inspect verbose output and make the command contract explicit.

pytest does not run on a documentation-only commit

If tests are part of every commit's local gate, use `always_run: true` with `pass_filenames: false`. If the project intentionally runs tests only on Python changes, document that policy and rely on CI for the authoritative full gate.

A formatter keeps failing on every run

Inspect the diff and tool order. A lint fix after formatting may rewrite code back into a shape that needs another format pass. Put fixers before formatters, then ensure the second run is idempotent.

Checkpoint: interpret hook results

Respond to hook evidence

Choose the next action that preserves staged-state and diagnostic evidence.

1. Ruff format reports that files were modified. What should happen next?

- **A.** Use –no-verify
- **B.** Delete the formatter hook
- **C.** Commit the old staged snapshot
- **D.** Review the diff, stage the intended formatting, and rerun the hooks

2. A YAML hook says `(no files to check) Skipped`. What does that mean?

- **A.** No selected staged file matched that hook's filters
- **B.** The whole repository is valid YAML
- **C.** The hook was deleted
- **D.** CI has already passed

3. Why run `pre-commit run --all-files` when introducing a hook?

- **A.** To create a commit automatically
- **B.** To apply the new policy to existing tracked files, not only the next staged patch
- **C.** To bypass file filters
- **D.** To push the branch

Write your choices before checking the answer key at the end of this book.

5. Keep hooks fast, visible, and bypassable

A hook that takes twenty minutes encourages people to avoid it. Put fast formatting, linting, and focused type checks early. Whether the full test suite belongs in pre-commit or pre-push depends on project size. The Midnight Museum suite is small enough to run locally on each commit.

Git permits bypassing verification with `git commit --no-verify`. This is useful for rare recovery workflows and is also why a local hook cannot be the shared authority. A contributor may not install it, may have a different environment, or may bypass it. CI repeats the committed contract on a clean shared runner.

Do not describe pre-commit as a security boundary. It is a feedback and consistency tool running code defined by a repository. Review hook sources and revisions before executing an unfamiliar project's configuration.

Update revisions as real dependency changes

For remote hooks:

```bash
pre-commit autoupdate
pre-commit run --all-files
```

Inspect the revision diff and tool behavior. An automatic version edit is a proposal, not evidence that the new version fits the project. Keep the update in a focused patch when possible.

Never erase user work to satisfy a hook

A formatter or fixer should touch only intended files. Inspect `git status` and `git diff`; do not reset, clean, or overwrite unrelated working-tree changes. :::

6. Build one named local quality contract

A project is easier to automate when the commands are stable and visible:

```text
ruff check src tests
ruff format --check src tests
mypy src
pytest -q
```

The pre-commit configuration may use a rewriting `ruff format` for convenience, while the final manual/CI gate uses `ruff format --check`. Keep that distinction explicit:

- **local repair loop:** tools may make reviewable changes;
- **acceptance gate:** commands report nonzero rather than modifying committed source.

Lesson 6 will place the check-only sequence in CI.

Checkpoint: limits of a local gate

Know what the local gate proves

Choose the conclusion supported by the local hook design.

1. Why does a passing pre-commit hook not replace CI?

- **A.** Hooks cannot run Python
- **B.** CI never runs commands
- **C.** A local hook can be absent, bypassed, or affected by uncommitted machine state
- **D.** Git forbids hooks in projects

2. Which command belongs in a check-only acceptance gate?

- **A.** ruff format src tests
- **B.** ruff check –fix

- **C.** git add -A
- **D.** ruff format –check src tests

3. After `pre-commit autoupdate` changes revisions, what provides meaningful acceptance evidence?

- **A.** Review the revision diff and rerun all hooks and project checks
- **B.** Assume latest always passes
- **C.** Delete the lock file
- **D.** Hide the changed YAML

Write your choices before checking the answer key at the end of this book.

7. Lab: build the museum project’s local gate

In the Midnight Museum Git project:

1. add local hooks for Ruff lint, Ruff format, MyPy, and pytest;
2. ensure project-wide tools do not receive appended staged filenames;
3. install the hook;
4. create and stage an unsorted import plus an unused import;
5. commit or run the hook and capture the first modified/failed output;
6. inspect, stage, and rerun the repaired file;
7. introduce a real MyPy argument mismatch and explain its diagnostic;
8. repair it rather than suppressing it; and
9. finish with a clean `pre-commit run --all-files` transcript.

Hint A: use the verified local order

Place Ruff lint before Ruff format, then MyPy and pytest. Use `language: system` for this lab because the active project environment already owns the tools.

Hint B: select the full-project commands

Set `pass_filenames: false` for `mypy src` and `pytest -q`. Add `always_run: true` to pytest if it should run on every commit.

Hint C: prove the second run is clean

After a hook modifies a file, inspect `git diff`, stage the intended result, and run `pre-commit run --all-files` again. A single modifying run is not completion.

Show a complete local configuration

YAML

```yaml
repos:
  - repo: local
    hooks:
      - id: ruff-check
        name: Ruff lint
        entry: ruff check
        language: system
```

YAML (CONTINUED)

```yaml
        types: [python]
      - id: ruff-format
        name: Ruff format
        entry: ruff format
        language: system
        types: [python]
      - id: mypy
        name: MyPy
        entry: mypy src
        language: system
        pass_filenames: false
      - id: pytest
        name: pytest
        entry: pytest -q
        language: system
        pass_filenames: false
        always_run: true
```

A clean run should end with four `Passed` statuses. Preserve the earlier modified-file and MyPy failure records too; they demonstrate that the gate actually detected and helped repair intended problems.

Key points

- Git hooks are local event scripts; pre-commit shares and manages their configuration and environments.
- Ordinary commit hooks check the staged snapshot, while `--all-files` establishes a repository-wide baseline.
- Passed, skipped, modified, and failed outputs carry different evidence.
- Review and stage tool modifications, then rerun until the gate is idempotent.
- Use explicit filters, `pass_filenames`, ordering, and environment choices so commands run in the intended scope.
- A fast local hook shortens feedback but remains bypassable and machine-local; CI is the shared clean gate.

References

- pre-commit official documentation
- Git hooks documentation
- Ruff integration with pre-commit

Continue to Lesson 6: Repeat the Quality Gate in CI

Repeat the Quality Gate in CI

Online lesson and runnable notebook

Build and diagnose a GitHub Actions workflow that installs a clean project and repeats the same check-only quality commands used locally.

- **Level:** Python Foundations
- **Estimated time:** 3–4 hours
- **You will learn:** Explain a workflow from event to exit status, keep local and CI commands aligned, and trace a failed clean run to source, setup, or configuration.
- **Practice in:** A local project plus supplied GitHub Actions files and run transcripts

1. A clean runner tests committed assumptions

The local museum gate passes. That result depends on one checkout, environment, and set of installed tools. Continuous integration starts a new automated run from committed files and a declared workflow.

Before writing YAML, answer these questions:

1 Which repository event should start the quality check?

2 What must the runner install before it can run project commands?

3 How does a failed command stop the job?

4 Which local and CI commands must remain identical?

5 What hidden local assumption can a clean environment expose?

Create `.github/workflows/quality.yml`:

YAML

```yaml
name: Quality

on:
  pull_request:
  push:

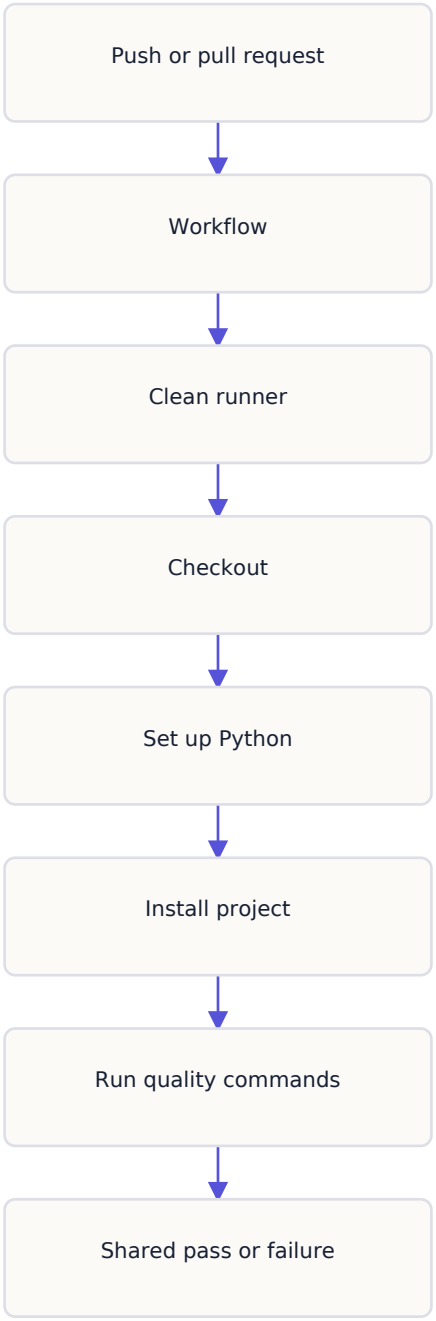
jobs:
  quality:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-python@v5
        with:
          python-version: "3.10"
      - name: Install project and quality tools
        run: python -m pip install -e ".[dev]"
      - name: Lint
        run: ruff check src tests
      - name: Check formatting
        run: ruff format --check src tests
      - name: Check types
        run: mypy src
      - name: Run tests
        run: pytest -q
```

The file describes one workflow named **Quality**:

- `on` declares events;
- `jobs` contains independently scheduled work;
- `quality` is a machine ID for one job;
- `runs-on` chooses a hosted runner image;
- `steps` execute in order within that job;
- `uses` invokes a reusable action; and
- `run` invokes a shell command.

A real project must declare its development tools in a way the install command can resolve. If the project uses Poetry, a lock file, or another environment manager, use its documented install command rather than copying `.[dev]` blindly.

A Git event creates a clean runner that reconstructs the project before executing the gate.



2. Order steps by their dependencies

The linter cannot inspect source before checkout. Tests cannot import the project before installation. Put each prerequisite before its consumer:

1. check out the commit/ref associated with the run;
2. select a supported Python interpreter;
3. install the declared project and development dependencies;
4. run fast static checks;
5. run the behavior suite; and

6 build documentation/package only if those artifacts belong to this gate.

A workflow run uses the workflow version stored in the associated commit or Git reference. Uncommitted local fixes do not reach it. That is a feature: CI asks whether committed state is self-contained.

A nonzero exit status is the gate signal

Ruff, MyPy, and pytest exit zero on success and nonzero on a detected problem or execution failure. GitHub Actions marks a normal `run` step failed when its command exits nonzero.

This command destroys that signal:

```bash
pytest -q || true
```

If pytest exits 1, the shell runs `true`, which exits 0. The step looks green while tests are red. Likewise, an accidental `continue-on-error: true` changes a required gate into informational output.

Use non-blocking steps only when failure is intentionally advisory and clearly labeled. This unit's quality commands are gates, so let them fail.

Checkpoint: workflow anatomy

Trace the workflow from event to result

Choose the field or step that supplies the stated evidence.

1. What does `runs-on: ubuntu-latest` select?

- **A.** The Python package version
- **B.** The runner environment for the job
- **C.** The branch to merge
- **D.** The MyPy target

2. Why must checkout and installation precede `pytest -q`?

- **A.** YAML requires alphabetical steps
- **B.** pytest installs Git
- **C.** The test step needs committed source and declared dependencies available on the runner
- **D.** GitHub ignores later steps

3. Why is `pytest -q || true` unsuitable for a required quality gate?

- **A.** It runs pytest twice
- **B.** It deletes tests
- **C.** It checks only one file
- **D.** It turns a failing pytest status into shell success

Write your choices before checking the answer key at the end of this book.

3. Make local and CI commands one contract

Drift creates confusing states:

Local hook	CI	Resulting risk
`ruff check src`	`ruff check src tests`	Test lint failures appear only after push
`mypy src`	no MyPy step	CI accepts type gaps local contributors reject
pytest 9 from lock	unbounded latest pytest	Same commit can receive different behavior
formatter rewrites	formatter check-only	Fine when the distinction is documented

Write the authoritative check-only sequence in the README, task runner, or a small script that both environments call. Direct commands are easiest for this small project:

```text
ruff check src tests
ruff format --check src tests
mypy src
pytest -q
```

The local pre-commit repair loop may run `ruff format` in place. The final local verification and CI must use `--check`. That is intentional behavior, not drift.

Pin enough state to reproduce the result

A clean environment still changes over time if it installs unconstrained latest tools. Use the project's chosen dependency record and review updates. This can be a lock file, exact constraints, or a controlled range plus regular testing.

Actions also have versions. A tag such as `actions/checkout@v4` tracks a major release. Security-sensitive projects may pin an immutable commit SHA and use an update process. Foundations learners need to recognize that an action is executable dependency code, not memorize one universal pinning policy.

Do not duplicate policy in the workflow

Keep rule selection, target Python, source paths, and MyPy strict settings in `pyproject.toml`. The workflow should call tools; it should not contain a second copy of every rule flag. One committed configuration reduces disagreement among editor, local command, hook, and CI.

4. Classify the failing stage before editing source

A red workflow is not always a code defect. Start at the earliest failed step and classify it.

Source-quality failure

```text
Run ruff check src tests
F821 Undefined name `rank_attempt`
Found 1 error.
Error: Process completed with exit code 1.
```

Reproduce locally from the project root:

```bash
ruff check src tests
```

Read the rule and repair the source or a justified policy boundary.

Installation failure

```text
Run python -m pip install -e ".[dev]"
ERROR: Directory '.' is not installable. Neither 'setup.py' nor 'pyproject.toml' found.
```

Do not change Python logic. The runner is in the wrong directory or the project metadata was not committed. Compare checkout paths and `working-directory`.

Environment/version failure

```text
SyntaxError: Pattern matching is only supported in Python 3.10 and greater
```

If a matrix job selected Python 3.9 while the project requires 3.10, align the matrix with the declared support range. If the project promises 3.9, the source is incompatible. The contract determines the repair.

Configuration failure

A malformed YAML file may prevent the workflow from starting, so no shell step exists to inspect. Use an editor/YAML parser, compare indentation, and read the workflow annotation. A correctly parsed workflow can still contain a wrong key or expression; consult the provider’s syntax reference.

Test failure only on CI

Compare:

- Python and dependency versions;
- operating system and path case;
- locale, timezone, environment variables, and clock;
- files present in Git versus only in the local working tree; and
- test order or leaked global state.

Do not label CI “random.” State the observed difference and reduce it through a controlled local reproduction or a more explicit fixture.

Checkpoint: diagnose the clean run

Find the earliest useful CI evidence

Classify the shown failure before choosing a change.

1. Installation says no pyproject.toml was found. What should you inspect first?

- **A.** The test assertion values

- **B.** Ruff quote style
- **C.** The checkout, committed files, and step working directory
- **D.** The leaderboard tie rule

2. A test passes locally but fails on CI. Which investigation is strongest?

- **A.** Rerun until green
- **B.** Compare interpreter, dependencies, OS paths, environment, committed files, and leaked state
- **C.** Disable the test
- **D.** Add `|| true`

3. Why start with the earliest failed workflow step?

- **A.** Later steps always run first
- **B.** GitHub sorts errors alphabetically
- **C.** Later failures or skips can be consequences of an unmet earlier dependency
- **D.** Only the first step has logs

Write your choices before checking the answer key at the end of this book.

5. Use a small version matrix for a real support claim

A library that claims Python 3.10–3.13 support should execute tests on those versions. A matrix avoids copying the job:

YAML

```yaml
jobs:
  quality:
    runs-on: ubuntu-latest
    strategy:
      fail-fast: false
      matrix:
        python-version: ["3.10", "3.11", "3.12", "3.13"]
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-python@v5
        with:
          python-version: ${{ matrix.python-version }}
      - name: Install project
        run: python -m pip install -e ".[dev]"
      - name: Run quality gate
        run: |
          ruff check src tests
          ruff format --check src tests
          mypy src
          pytest -q
```

This produces one job variant per value. `fail-fast: false` lets all variants report, which helps distinguish “source fails everywhere” from “only the oldest supported interpreter fails.”

Do not add operating systems, dependency combinations, and dozens of versions without a support question. Every matrix dimension multiplies cost and output. The Midnight Museum exercise can start with Python 3.10; the course repository’s real CI uses a supported-version matrix.

Separate steps or one gate step?

Separate named steps show exactly which tool failed and can make logs easier to scan. One script reduces command duplication if local and CI call the same script. Both are valid:

- for a small lesson, separate steps reveal the chain;
- for a mature project, one versioned `quality` task can keep environments aligned while still printing each command.

Choose the design that makes the contract and failure visible.

6. State what a passing workflow proves

A green workflow demonstrates that the workflow version associated with the run completed its selected jobs and steps successfully in their declared runner environments. It does not prove:

- every possible input works;
- every behavior has a test;
- a security vulnerability is absent;
- a contributor ran local hooks;
- production matches the runner; or
- the code is easy to maintain.

CI reduces dependence on hidden local state and gives collaborators a shared result. Its strength comes from a clear, relevant gate—not from the color green alone.

Checkpoint: CI scope and design

Design a defensible shared gate

Choose the workflow conclusion or design that matches the support claim.

1. What does a green workflow demonstrate?

- **A.** No bug exists
- **B.** Every user environment works
- **C.** All future dependency versions work
- **D.** The selected committed workflow completed successfully in its declared runner environments

2. When is a Python-version matrix justified?

- **A.** When the project claims support for several versions and needs execution evidence for each
- **B.** Whenever a file has more than one function
- **C.** To replace local tests
- **D.** To format code differently by version

3. Where should Ruff rule selection and MyPy strict settings normally live?

- **A.** Only in one developer’s editor
- **B.** In committed project configuration such as pyproject.toml, which local and CI commands share

- **C.** Repeated independently in every workflow line
- **D.** In test expected values

Write your choices before checking the answer key at the end of this book.

7. Lab: make the remote gate match the local gate

Start with this incomplete workflow:

```yaml
name: Quality

on:
  pull_request:

jobs:
  quality:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-python@v5
        with:
          python-version: "3.10"
      - name: Install
        run: python -m pip install -e ".[dev]"
      - name: Quality
        run: |
          ruff check src || true
          pytest -q
```

Complete these tasks:

1. remove the hidden-success behavior;
2. make Ruff inspect `src` and `tests`;
3. add Ruff format check, strict MyPy, and pytest;
4. keep project policy in `pyproject.toml`;
5. name steps so the first failed tool is visible;
6. compare each CI command with the final local command sequence;
7. explain how the workflow would reveal an uncommitted local fixture; and
8. diagnose three supplied cases: missing metadata, one Python-version failure, and a Ruff rule failure.

If you have a GitHub repository, you can push a branch and inspect the real run. If not, retain the authored YAML, parse it locally, and annotate the supplied transcripts. The assessed artifact and reasoning are the same.

Hint A: preserve exit status

Remove `|| true`. Put each required command in a normal `run` step so a nonzero status fails the job.

Hint B: use check-only formatting

CI should call `ruff format --check src tests`; it should not rewrite runner files that cannot be committed back to the branch.

Hint C: compare the four commands

The intended sequence is Ruff lint, Ruff format check, MyPy on `src`, and pytest. Ensure the same paths appear in the local acceptance gate.

Show one complete workflow

YAML

```yaml
name: Quality

on:
  pull_request:
  push:

jobs:
  quality:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-python@v5
        with:
          python-version: "3.10"
      - name: Install project and quality tools
        run: python -m pip install -e ".[dev]"
      - name: Lint
        run: ruff check src tests
      - name: Check formatting
        run: ruff format --check src tests
      - name: Check types
        run: mypy src
      - name: Run tests
        run: pytest -q
```

The install command must match the project's actual dependency declaration. A passing local gate immediately before push and this clean runner result provide two complementary environment checks.

Key points

- CI reconstructs committed project state on a declared runner, exposing hidden local files, dependencies, paths, and environment assumptions.
- Order checkout, interpreter setup, installation, and checks by dependency.
- Required commands must preserve nonzero exit status; do not hide failures with `|| true` or accidental non-blocking settings.
- Keep local and CI check-only commands aligned and project policy in committed configuration.
- Diagnose the earliest failed stage as source, configuration, installation, or environment evidence before editing application code.
- A version matrix should test a real support claim, and a green result remains bounded by the workflow that actually ran.

References

- GitHub Actions workflow concepts
- GitHub Actions workflow syntax
- GitHub Actions: Add scripts to a workflow
- GitHub Actions: Using a matrix for jobs

Continue to Lesson 7: Refactor in Small, Verified Steps

Refactor in Small, Verified Steps

Online lesson and runnable notebook

Improve names, functions, conditionals, duplication, and side-effect boundaries through small diffs that preserve checked behavior.

- **Level:** Python Foundations
- **Estimated time:** 4–5 hours
- **You will learn:** Name a concrete maintenance cost, make one structural change, and use tests, types, lint, formatting, and diffs to show that observable behavior stayed stable.
- **Practice in:** The complete Midnight Museum Git project and quality gate

1. Refactoring changes structure on purpose

The Midnight Museum scorer works and its tests pass. A new requirement is coming: different galleries will apply different bonus rules. The current function combines validation, scoring, sorting, and text output:

PYTHON

```
def build_board(attempts):
    rows = []
    for attempt in attempts:
        if not attempt["team"].strip():
            raise ValueError("team cannot be empty")
        if not 0 <= attempt["clues_found"] <= 8:
            raise ValueError("clues_found must be between 0 and 8")
        score = attempt["clues_found"] * 30
        score += 25 if attempt["bonus_clue"] else 0
        score -= max(0, attempt["elapsed_minutes"] - 45) * 2
        rows.append((max(0, score), attempt["team"]))
    rows.sort(reverse=True)
    return "\n".join(f"{team}: {score}" for score, team in rows)
```

Before editing, answer:

1. Which observable behaviors already belong to the contract?
2. What specific maintenance task is expensive in the current structure?
3. Which smallest structural move would reduce that cost?
4. Which focused check should run immediately after the move?
5. Which final gate and diff will show the complete result?

Refactoring changes internal structure without intentionally changing observable behavior. These are not the same activity:

- changing the bonus from 25 to 40 is a behavior change;
- repairing a wrong tie order is a bug fix;
- formatting layout is a mechanical style pass;
- renaming a private local without changing output is a refactor; and
- replacing the whole project from memory is a rewrite, not a sequence of controlled refactors.

Separate behavior changes from structural changes when possible. A reviewer can then ask one clear question of each patch.

2. Freeze the behavior you intend to preserve

Start from a known commit and run the complete gate:

```bash
git status --short
pytest -q
ruff check src tests
ruff format --check src tests
mypy src
```

Write down the exact result. Then identify behavior that matters to callers:

- public function names and accepted arguments;
- returned values and types;
- exception types and messages promised by the interface;
- output text and ordering;
- whether caller-owned collections are mutated;
- file, terminal, or collaborator side effects; and
- performance only when it is part of a real requirement.

A characterization test records important existing behavior when the original intent is unclear. It is not an endorsement of the behavior; it creates a safety boundary for structural work. If the behavior is wrong, change it later with an explicit new requirement and test.

```python
def test_build_board_preserves_current_tie_order() -> None:
    attempts = [
        {"team": "Amber", "clues_found": 2, "elapsed_minutes": 40, "bonus_clue": False},
        {"team": "Blue", "clues_found": 2, "elapsed_minutes": 40, "bonus_clue": False},
    ]
    assert build_board(attempts) == "Blue: 60\nAmber: 60"
```

The descending team tie order may be surprising. Preserve it during the refactor. A later product decision can change it to alphabetical ascending with a separate failing test and release note.

Tests protect selected behavior, not intention by magic

A weak suite may miss input mutation:

```python
def test_board_text() -> None:
    attempts = [
        {"team": "Moon Moths", "clues_found": 6, "elapsed_minutes": 42, "bonus_clue": True},
    ]
    assert build_board(attempts) == "Moon Moths: 205"
```

Add the ownership boundary if callers rely on it:

```python
from copy import deepcopy


def test_build_board_does_not_mutate_attempts() -> None:
    attempts = [
        {"team": "Moon Moths", "clues_found": 6, "elapsed_minutes": 42, "bonus_clue": True},
    ]
    original = deepcopy(attempts)
    build_board(attempts)
    assert attempts == original
```

Do not write assertions for every private line. Protect the behavior a caller or collaborator can observe.

Checkpoint: define the change boundary

Separate structure from behavior

Choose the classification or evidence that matches the proposed change.

1. Changing the bonus from 25 to 40 is primarily what kind of work?

- **A.** Formatting
- **B.** Import sorting
- **C.** A behavior/requirement change
- **D.** A private rename

2. Why add a characterization test for a surprising current tie order before refactoring?

- **A.** To prove the order is ideal
- **B.** To make sorting faster
- **C.** To replace the requirements
- **D.** To detect an accidental structural change while the product decision remains separate

3. Which check detects accidental mutation of caller-owned attempts?

- **A.** Copy input before the call and assert it is equal afterward
- **B.** Count source lines
- **C.** Run only Ruff format
- **D.** Assert the output is a string

Write your choices before checking the answer key at the end of this book.

3. Make one named move and stop

A safe rhythm is deliberately small:

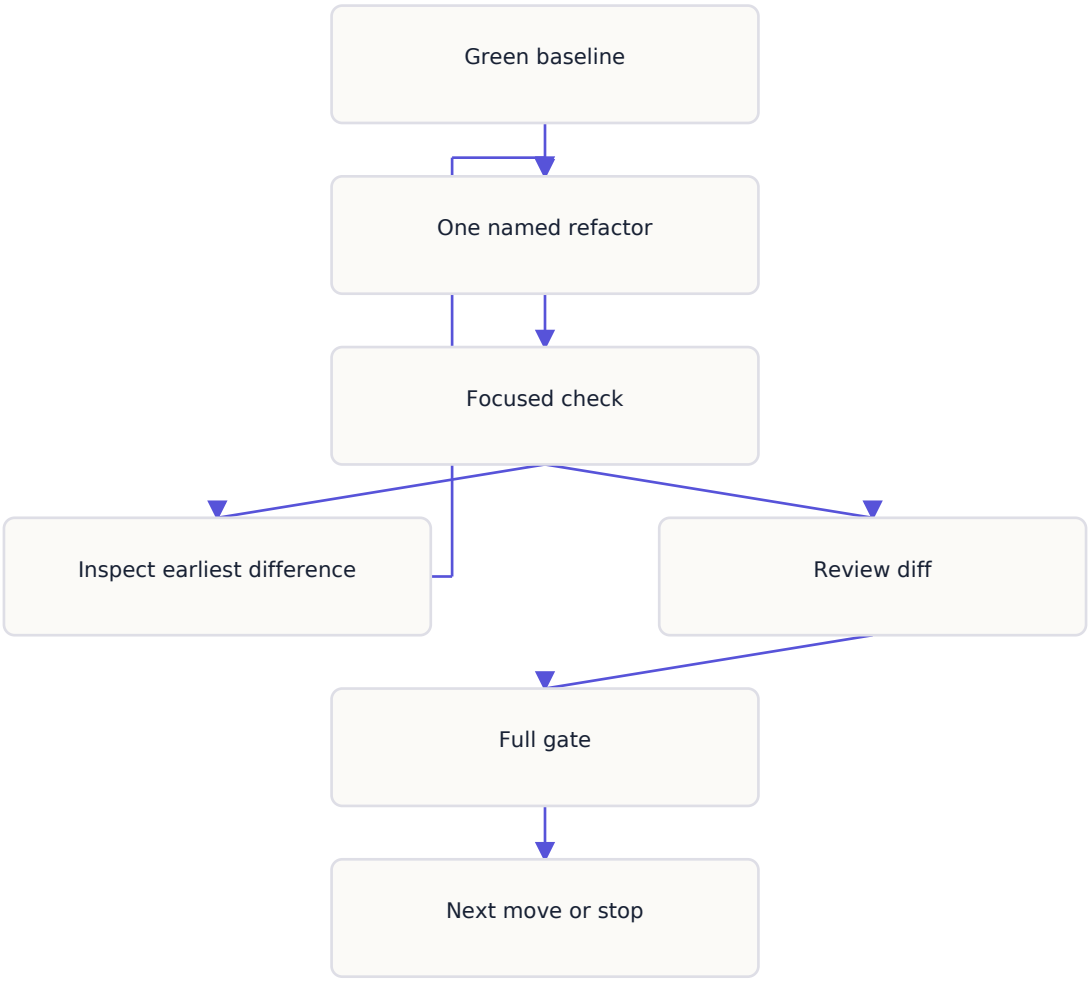
1. state the maintenance problem;
2. choose one refactoring move;
3. make only that move;
4. run the smallest relevant check;

5 inspect the diff;

6 run the broader gate at a meaningful checkpoint; and

7 commit or continue only when the state is understood.

Each verified move returns to a known state before another structural decision begins.



Focused check → Inspect earliest difference: Fail

Focused check → Review diff: Pass

Rename for the represented concept

Rename a private variable and all its uses together:

```python
def overtime_penalty(elapsed_minutes: int) -> int:
    overtime_minutes = max(0, elapsed_minutes - 45)
    return overtime_minutes * 2
```

Avoid search-and-replace across unrelated `time` names. Editor rename tools and static checks help, but inspect the diff to ensure strings, docs, public names, or other scopes did not change accidentally.

Extract a named intermediate value

This expression is correct but makes three rules compete:

```python
score = max(0, attempt["clues_found"] * 30 + (25 if attempt["bonus_clue"] else 0) - max(0, attempt["ela ↪
    psed_minutes"] - 45) * 2)
```

Expose the stages:

```python
clue_points = attempt["clues_found"] * 30
bonus_points = 25 if attempt["bonus_clue"] else 0
overtime_minutes = max(0, attempt["elapsed_minutes"] - 45)
overtime_penalty = overtime_minutes * 2
score = max(0, clue_points + bonus_points - overtime_penalty)
```

The goal is not more locals. The next bonus/overtime change now has named locations and separately inspectable values.

Extract a cohesive function

After named stages are stable, extract scoring:

```python
def score_attempt(attempt: Attempt) -> int:
    clue_points = attempt["clues_found"] * 30
    bonus_points = 25 if attempt["bonus_clue"] else 0
    overtime_minutes = max(0, attempt["elapsed_minutes"] - 45)
    overtime_penalty = overtime_minutes * 2
    return max(0, clue_points + bonus_points - overtime_penalty)
```

Keep validation either inside this function by contract or in a clear boundary function; do not accidentally remove it during extraction. Run focused scoring and invalid-input tests immediately.

Introduce a named constant when the name explains policy

```python
BONUS_POINTS = 25
OVERTIME_START_MINUTES = 45
OVERTIME_POINTS_PER_MINUTE = 2
```

A constant is valuable when the name exposes why a number matters or several uses must stay synchronized. Replacing every `0`, `1`, and `2` with a constant can make ordinary operations harder to read.

4. Simplify decisions without changing their edges

Nested code can often become guard clauses:

```python
def validate_attempt(attempt: Attempt) -> None:
    if not attempt["team"].strip():
        raise ValueError("team cannot be empty")
    if not 0 <= attempt["clues_found"] <= 8:
        raise ValueError("clues_found must be between 0 and 8")
    if attempt["elapsed_minutes"] < 0:
        raise ValueError("elapsed_minutes cannot be negative")
```

The order is observable when more than one field is invalid because the first exception wins. Preserve that order unless a behavior change is intended.

Replace duplication only after identifying what is truly shared

These branches repeat text shape:

```python
def badge_message(position: int, team: str) -> str:
    if position == 1:
        return f"Gold badge for {team}"
    if position == 2:
        return f"Silver badge for {team}"
    return f"Blue badge for {team}"
```

Extract the changing value:

```python
def badge_message(position: int, team: str) -> str:
    if position == 1:
        color = "Gold"
    elif position == 2:
        color = "Silver"
    else:
        color = "Blue"
    return f"{color} badge for {team}"
```

A lookup table can be clearer for stable data:

```python
BADGE_BY_POSITION = {1: "Gold", 2: "Silver"}


def badge_message(position: int, team: str) -> str:
    color = BADGE_BY_POSITION.get(position, "Blue")
    return f"{color} badge for {team}"
```

Do not force conditions into data when the branches perform distinct behavior or need different validation. Compare the next likely change: adding a third special color is simple in the table; adding a branch with a notification side effect may deserve explicit control flow.

Shorter can be less clear

This one-liner combines validation, defaulting, selection, and formatting:

```python
def label(record):
    return f"{record.get('team', '').strip() or 'UNKNOWN'}:{max(0, record.get('score', 0))}"
```

A few named statements give failures and decisions visible homes. Refactoring should reduce reasoning cost, not win a line-count contest.

Checkpoint: choose a useful move

Choose one structural improvement

Choose the move that addresses the stated maintenance cost without changing the contract.

1. Bonus and overtime rules change independently from report wording. Which refactor creates the clearest boundary?

- **A.** Put every statement on one line
- **B.** Extract `score_attempt` and keep rendering separate
- **C.** Delete intermediate names
- **D.** Change the public output

2. When does `OVERTIME_START_MINUTES = 45` improve the code most?

- **A.** Whenever any integer appears
- **B.** Because constants execute faster
- **C.** When the name exposes policy/units or synchronized uses of the threshold
- **D.** When tests are absent

3. Why preserve guard-clause order during a behavior-preserving extraction?

- **A.** Python requires alphabetical fields
- **B.** Ruff sorts conditions
- **C.** MyPy cannot check exceptions
- **D.** For input with several invalid fields, the first raised exception is observable

Write your choices before checking the answer key at the end of this book.

5. Separate pure calculation from effects

A function that reads a file, calculates scores, prints, and writes output is hard to check without arranging all effects. Move boundaries outward:

```python
from pathlib import Path


def build_report(attempts: list[Attempt]) -> str:
    ranked = rank_attempts(attempts)
    return render_leaderboard(ranked)


def write_report(input_path: Path, output_path: Path) -> None:
    attempts = load_attempts(input_path)
    report = build_report(attempts)
    output_path.write_text(report, encoding="utf-8")
```

Now `build_report` is a pure value transformation. `write_report` owns file interaction. This is a structural improvement only if paths, encoding, output, exceptions, and call behavior remain compatible with the existing contract.

Refactors can move side effects accidentally

Compare:

```python
def announce_and_score(attempt: Attempt) -> int:
    print(f"Scoring {attempt['team']}")
    validate_attempt(attempt)
    return score_attempt(attempt)
```

Moving validation before printing changes output for invalid input. That may be a better design, but it is observable. Decide explicitly and test the intended order rather than calling it an invisible cleanup.

Public names are contracts; private names are freer

Renaming `_calculate_bonus` is usually internal. Renaming `score_attempt` breaks imports and callers unless the project updates them or provides a migration. This course is pre-release and can approve route/ID resets, but an application API still needs an intentional contract decision. A public rename is not a behavior-preserving local refactor merely because tests were edited at the same time.

6. Use the diff as the review artifact

After each move:

```bash
git diff --stat
git diff -- src/midnight_museum/quest.py
git diff --check
```

Ask:

- Does the diff contain one explainable structural purpose?
- Did a formatter rewrite unrelated files?
- Did expected values change without a requirement change?
- Did public names, exceptions, ordering, mutation, or output move?
- Are new abstractions named from the domain rather than implementation trivia?
- Can the old and new behavior be compared with the same tests?

Then run the gate:

```bash
ruff check src tests
ruff format --check src tests
mypy src
pytest -q
```

A small diff plus a green gate is strong evidence, not certainty. Explain what the tests and checks cover and what remains a judgment.

Know when to stop

An abstraction has a cost: another name, file, call, and navigation step. Stop when the present maintenance problem is solved and the code has a coherent shape. Do not add a strategy class, plugin registry, or generic framework for a single fixed bonus rule. Future requirements may arrive differently from the one you imagined.

Checkpoint: review and stopping

Review the refactor as evidence

Choose the check or stopping rule that keeps the patch defensible.

1. What makes a refactoring diff easiest to review?

- **A.** It combines a feature, tool upgrade, and rename
- **B.** It changes every file
- **C.** It removes all tests
- **D.** It has one named structural purpose and no unexplained expected-value changes

2. Moving validation before a print changes output for invalid input. How should that be treated?

- **A.** As invisible formatting
- **B.** As an observable behavior decision with its own requirement/check
- **C.** As proof that print is wrong
- **D.** As a type-only change

3. When should you stop extracting abstractions?

- **A.** Only after every line is a function
- **B.** When no constants remain
- **C.** When the stated maintenance problem is solved and another layer would add indirection without present value
- **D.** Never; more layers are always clearer

Write your choices before checking the answer key at the end of this book.

7. Lab: refactor the museum ranking pipeline without a rewrite

Use the complete `build_board` function from Section 1 and the existing green tests. Preserve:

- public name `build_board`;
- exact report text;
- zero score floor;
- validation exception types/messages/order;
- descending score and descending team tie order;
- no mutation of input; and
- Python 3.10-compatible typed interfaces.

Perform these moves separately:

1. add missing characterization checks for tie order and mutation;
2. extract named score stages;

3 extract `validate_attempt` without changing exception order;

4 extract `score_attempt` and rerun only scoring/invalid tests;

5 introduce only constants whose names expose policy;

6 extract ranking and rendering boundaries;

7 annotate the completed private/public boundaries honestly;

8 run Ruff and review its formatting separately;

9 inspect the full diff for an accidental behavior edit; and

10 run the complete local gate.

Keep a change log:

Move	Maintenance problem	Focused check	Diff observation
Add tie test	Current order was implicit	tie test passes	No production edit
Extract scoring	Bonus rule mixed with report	scoring tests pass	Public output unchanged
...	...	...	...

Hint A: freeze unusual behavior first

Use the descending `Blue` then `Amber` tie example and a deep copy of attempts. Do not “improve” the tie policy during this structural lab.

Hint B: extract in two stages

First create named intermediate scoring values inside `build_board`. When the focused tests pass, move those same statements into `score_attempt`. This makes a failed extraction easier to compare.

Hint C: preserve tuple sorting exactly

`ranked.sort(reverse=True)` on `(score, team)` tuples implements descending order for both fields. Keep that operation if exact current behavior is the contract. A key using `(-score, team)` would change ties.

Show one complete behavior-preserving result

PYTHON

```
from collections.abc import Sequence
from typing import TypedDict

CLUE_POINTS = 30
BONUS_POINTS = 25
OVERTIME_START_MINUTES = 45
OVERTIME_POINTS_PER_MINUTE = 2


class Attempt(TypedDict):
    team: str
    clues_found: int
    elapsed_minutes: int
    bonus_clue: bool


def validate_attempt(attempt: Attempt) -> None:
    if not attempt["team"].strip():
        raise ValueError("team cannot be empty")
```

PYTHON (CONTINUED)

```
    if not 0 <= attempt["clues_found"] <= 8:
        raise ValueError("clues_found must be between 0 and 8")


def score_attempt(attempt: Attempt) -> int:
    validate_attempt(attempt)
    clue_points = attempt["clues_found"] * CLUE_POINTS
    bonus_points = BONUS_POINTS if attempt["bonus_clue"] else 0
    overtime_minutes = max(
        0,
        attempt["elapsed_minutes"] - OVERTIME_START_MINUTES,
    )
    overtime_penalty = overtime_minutes * OVERTIME_POINTS_PER_MINUTE
    return max(0, clue_points + bonus_points - overtime_penalty)


def rank_attempts(attempts: Sequence[Attempt]) -> list[tuple[int, str]]:
    ranked = [
        (score_attempt(attempt), attempt["team"])
        for attempt in attempts
    ]
    ranked.sort(reverse=True)
    return ranked


def build_board(attempts: Sequence[Attempt]) -> str:
    return "\n".join(
        f"{team}: {score}" for score, team in rank_attempts(attempts)
    )
```

This solution intentionally preserves the original validation shown in Section 1: team and clue count. If the project contract also rejects negative elapsed minutes, add that as a separate behavior change with a failing example first. The constants and extra functions are justified by the planned bonus-rule change and independently checkable stages.

Key points

- Refactoring changes structure without intentionally changing observable behavior; features and bug fixes need separate requirements and evidence.
- Freeze important output, exceptions, ordering, mutation, and side effects before moving code.
- Make one named move, run a focused check, inspect the diff, and return to the full gate at useful checkpoints.
- Rename, extract values/functions, introduce policy constants, simplify branches, remove real duplication, and isolate effects only when they solve a stated maintenance problem.
- Shorter and more abstract are not synonyms for clearer.
- Stop when the problem is solved; every new layer has a reading and navigation cost.

References

- pytest: Good integration practices
- Python tutorial: Coding Style
- Git diff documentation
- Refactoring catalog

Continue to the Unit Challenge

Unit Challenge: Untangle the Clockwork Observatory

Online lesson and runnable notebook

Rescue a working but difficult signal-ranking project by preserving its report while adding clear structure, strict types, local hooks, and a reproducible CI gate.

- **Level:** Python Foundations · Unit 14 challenge
- **Estimated time:** 60–120 minutes
- **Outcome assessed:** Preserve a signal report while making its source, type contracts, tool policy, local gate, and CI workflow understandable and reproducible.
- **Evidence:** Passing progressive assertions, a clean check-only gate, one reviewed refactoring log, and an explained workflow.

1. Open the observatory without changing its signals

The Clockwork Observatory receives unusual sky signals. Its clockwork dome opens only for high-priority observations. The current script prints the correct sample report, but one large function mixes validation, scoring, classification, sorting, and text rendering. Names are vague, the quality configuration is missing, and no clean runner repeats the local result.

Your goal is to **untangle the project without changing the published report or boundary behavior**.

The supplied scoring rules are complete:

- `clarity` is an integer from 0 through 100 and contributes twice its value;
- a rare signal earns 25 bonus points;
- each minute of age subtracts one point, capped at a 60-point penalty;
- a score of 180 or more means `OPEN DOME`;
- a score from 120 through 179 means `REVIEW`;
- a lower score means `ARCHIVE`;
- ranking uses descending score, then ascending signal name;
- caller-provided records must not be mutated; and
- invalid name, clarity, and age values raise the exact errors supplied below.

Aim for a sequence of small rescues

Run the baseline first. Make one extraction or configuration change, run the smallest matching check, and record the result. The challenge is difficult because several quality signals must agree—not because you must invent hidden domain rules. :::

2. Understand the acceptance example

The observatory receives:

```python
signals = [
    {"name": "Lyra Echo", "clarity": 90, "age_minutes": 5, "rare": True},
    {"name": "Orion Bell", "clarity": 80, "age_minutes": 10, "rare": False},
    {"name": "Cygnus Key", "clarity": 80, "age_minutes": 10, "rare": False},
    {"name": "Moth Light", "clarity": 50, "age_minutes": 80, "rare": False},
]
```

The exact result is:

```text
CLOCKWORK OBSERVATORY
1. Lyra Echo | 200 | OPEN DOME
2. Cygnus Key | 150 | REVIEW
3. Orion Bell | 150 | REVIEW
4. Moth Light | 40 | ARCHIVE
```

Before editing, calculate all four scores by hand. Explain why `Cygnus Key` appears before `Orion Bell`, and predict the report for an empty sequence.

Project shape

Create this local project:

```text
clockwork-observatory/
├── pyproject.toml
├── .pre-commit-config.yaml
├── .github/
│   └── workflows/
│       └── quality.yml
├── src/
│   └── clockwork_observatory/
│       ├── __init__.py
│       └── signals.py
└── tests/
    └── test_signals.py
```

Use the notebook for calculation experiments if useful, but the hook and CI artifacts belong in a Git project.

3. Start from the contract

Place this valid but difficult implementation in `signals.py`. It establishes the baseline public report; you will replace its internal structure while keeping `build_observatory_report` public.

```python
from collections.abc import Sequence
from typing import TypedDict


class Signal(TypedDict):
    name: str
    clarity: int
    age_minutes: int
    rare: bool


def build_observatory_report(signals: Sequence[Signal]) -> str:
    """Return the complete ranked observatory report."""
    rows = []
```

PYTHON (CONTINUED)

```
    for item in signals:
        n = item["name"].strip()
        c = item["clarity"]
        a = item["age_minutes"]
        if not n:
            raise ValueError("signal name cannot be empty")
        if not 0 <= c <= 100:
            raise ValueError("clarity must be between 0 and 100")
        if a < 0:
            raise ValueError("age_minutes cannot be negative")
        s = c * 2 + (25 if item["rare"] else 0) - min(a, 60)
        if s >= 180:
            d = "OPEN DOME"
        elif s >= 120:
            d = "REVIEW"
        else:
            d = "ARCHIVE"
        rows.append({"name": n, "score": s, "decision": d})
    rows.sort(key=lambda row: (-row["score"], row["name"]))
    lines = ["CLOCKWORK OBSERVATORY"]
    for position, row in enumerate(rows, start=1):
        lines.append(
            f"{position}. {row['name']} | {row['score']} | {row['decision']}"
        )
    return "\n".join(lines)
```

Preserve these names and signatures in the completed design:

PYTHON

```
class RankedSignal(TypedDict):
    """A validated signal with its computed score and decision."""

    name: str
    score: int
    decision: str


def score_signal(signal: Signal) -> int:
    """Return priority points for one validated signal."""
    ...


def classify_signal(score: int) -> str:
    """Return the observatory decision for a signal score."""
    ...


def rank_signals(signals: Sequence[Signal]) -> list[RankedSignal]:
    """Return ranked copies without changing the caller's sequence."""
    ...


def build_observatory_report(signals: Sequence[Signal]) -> str:
    """Return the complete ranked observatory report."""
    ...
```

The ellipses are design placeholders for this displayed contract; replace them with working bodies in the project. Do not change the public names to make tests pass.

Quality configuration contract

Configure these policies in `pyproject.toml`:

- project requires Python 3.10 or newer;

- Ruff targets `py310`, line length 88, and `src`/`tests`;
- Ruff selects `E`, `F`, `I`, `UP`, `B`, and `RUF`;
- MyPy uses Python 3.10, strict mode, and visible error codes; and
- pytest adds `src` to its import path and discovers `tests`.

The final check-only command sequence is:

```text
ruff check src tests
ruff format --check src tests
mypy src
pytest -q
```

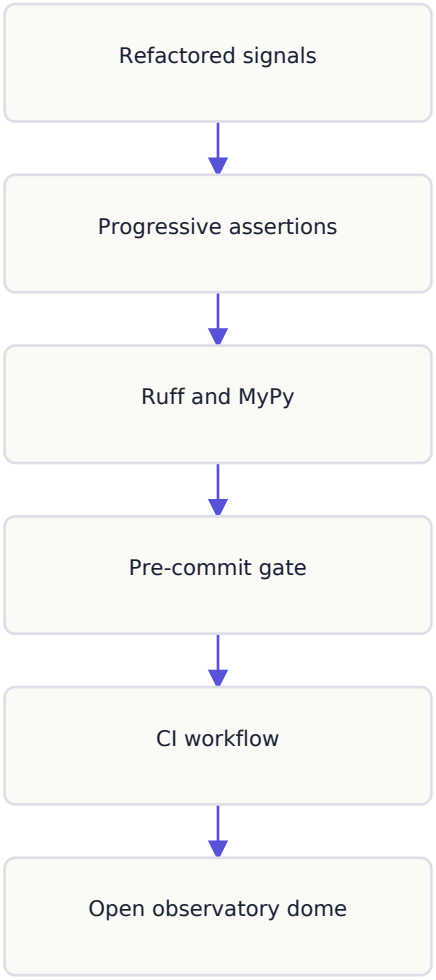
4. Build in small stages

Use this order to keep cause and evidence connected:

1. Copy the starter and run the report assertions as a green baseline.
2. Add `RankedSignal`; run MyPy and explain every diagnostic before repairing it.
3. Extract `score_signal`; rerun only score and validation checks.
4. Extract `classify_signal`; rerun threshold boundaries.
5. Extract `rank_signals`; verify tie order and no input mutation.
6. Reduce `build_observatory_report` to orchestration and rendering; compare the exact report.
7. Add Ruff/MyPy/pytest configuration and finish a clean manual gate.
8. Add local pre-commit hooks for all four commands. Use check-only formatting in the final gate.
9. Complete `quality.yml` so a clean GitHub Actions runner checks out source, sets up Python 3.10, installs the tools, and runs the same commands.
10. Inspect the final diff and explain why each extracted function has one reason to change.

Do not combine a scoring behavior change with the refactor. If you think a rule could be improved, record it as a future requirement after the challenge.

The dome opens only after the same source passes behavior, static, local, and clean-runner checks.



5. Run progressive assertions

Run these after the matching stage. A failing assertion identifies the earliest contract difference; do not edit expected values to manufacture a pass.

PYTHON

```
from copy import deepcopy

lyra = {
    "name": "Lyra Echo",
    "clarity": 90,
    "age_minutes": 5,
    "rare": True,
}
moth = {
    "name": "Moth Light",
    "clarity": 50,
    "age_minutes": 80,
    "rare": False,
}

assert score_signal(lyra) == 200
assert score_signal(moth) == 40
assert classify_signal(180) == "OPEN DOME"
assert classify_signal(179) == "REVIEW"
assert classify_signal(120) == "REVIEW"
```

PYTHON (CONTINUED)

```
assert classify_signal(119) == "ARCHIVE"
```

PYTHON

```
signals = [
    {"name": "Lyra Echo", "clarity": 90, "age_minutes": 5, "rare": True},
    {"name": "Orion Bell", "clarity": 80, "age_minutes": 10, "rare": False},
    {"name": "Cygnus Key", "clarity": 80, "age_minutes": 10, "rare": False},
    {"name": "Moth Light", "clarity": 50, "age_minutes": 80, "rare": False},
]
original = deepcopy(signals)
ranked = rank_signals(signals)

assert [item["name"] for item in ranked] == [
    "Lyra Echo",
    "Cygnus Key",
    "Orion Bell",
    "Moth Light",
]
assert signals == original
assert build_observatory_report([]) == "CLOCKWORK OBSERVATORY"
```

PYTHON

```
expected_report = """CLOCKWORK OBSERVATORY
1. Lyra Echo | 200 | OPEN DOME
2. Cygnus Key | 150 | REVIEW
3. Orion Bell | 150 | REVIEW
4. Moth Light | 40 | ARCHIVE"""
assert build_observatory_report(signals) == expected_report
```

Validation checks:

PYTHON

```
invalid_signals = [
    ({"name": " ", "clarity": 50, "age_minutes": 1, "rare": False}, "signal name cannot be empty"),
    ({"name": "Nova", "clarity": 101, "age_minutes": 1, "rare": False}, "clarity must be between 0 and 
    100"),
    ({"name": "Nova", "clarity": 50, "age_minutes": -1, "rare": False}, "age_minutes cannot be negative
    "),
]

for signal, expected_message in invalid_signals:
    try:
        score_signal(signal)
    except ValueError as error:
        assert str(error) == expected_message
    else:
        raise AssertionError("invalid signal was accepted")
```

Gate evidence targets

Your final record should show:

TEXT

```
All checks passed!
3 files already formatted
Success: no issues found in 2 source files
11 passed
```

Exact timing and file counts can differ if your test layout differs. The four commands must exit zero, and a second `pre-commit run --all-files` must make no new modification.

Expected values are the contract

Do not change `200`, tie order, exception messages, or the report to match an incorrect implementation. Trace the first mismatch to one scoring, validation, ranking, or rendering stage. :::

6. Use the hint ladder only when needed

Hint 1

Extract `score_signal` without changing a single expression first. Keep validation in the same order, name the three score components, and run only the score/invalid checks.

Hint 2

In `rank_signals`, calculate `score = score_signal(signal)` once, then construct `RankedSignal(name=..., score=score, decision=classify_signal(score))`. Return `sorted(ranked, key=lambda signal: (-signal["score"], signal["name"]))`.

Hint 3

Use local pre-commit hooks with `language: system`; set `pass_filenames: false` for `mypy src` and `pytest -q`. The CI workflow should use check-only commands and must not contain `|| true` or rewrite formatting.

7. Keep debugging evidence

Keep one real failure from the rescue. Good candidates include a reversed tie, a score calculated twice with inconsistent data, a MyPy TypedDict mismatch, a hook modification, or a CI step running from the wrong directory.

Failure	Evidence	Hypothesis	Controlled change	Verified rerun
What failed first?	Exact assertion, diagnostic, or hook line	Which single cause fits?	What one edit tested it?	Which focused and full checks now pass?

Also keep a refactoring log:

Stage	Structural purpose	Behavior check	Diff observation
Extract scoring	Isolate policy calculation	score/validation checks	Report untouched
Extract ranking	Expose tie and ownership rules	rank/mutation checks	Input still copied
Add gate	Reproduce policy	four zero exits	No tool rewrites remain

8. Inspect a complete solution only after your attempt

Show the complete source and quality configuration

A complete `signals.py` is:

```python
from collections.abc import Sequence
from typing import TypedDict
```

PYTHON (CONTINUED)

```
class Signal(TypedDict):
    """One signal received by the observatory."""

    name: str
    clarity: int
    age_minutes: int
    rare: bool


class RankedSignal(TypedDict):
    """A validated signal with its computed score and decision."""

    name: str
    score: int
    decision: str


def score_signal(signal: Signal) -> int:
    """Return priority points for one validated signal."""
    name = signal["name"].strip()
    clarity = signal["clarity"]
    age_minutes = signal["age_minutes"]

    if not name:
        raise ValueError("signal name cannot be empty")
    if not 0 <= clarity <= 100:
        raise ValueError("clarity must be between 0 and 100")
    if age_minutes < 0:
        raise ValueError("age_minutes cannot be negative")

    rarity_bonus = 25 if signal["rare"] else 0
    freshness_penalty = min(age_minutes, 60)
    return clarity * 2 + rarity_bonus - freshness_penalty


def classify_signal(score: int) -> str:
    """Return the observatory decision for a signal score."""
    if score >= 180:
        return "OPEN DOME"
    if score >= 120:
        return "REVIEW"
    return "ARCHIVE"


def rank_signals(signals: Sequence[Signal]) -> list[RankedSignal]:
    """Return ranked copies without changing the caller's sequence."""
    ranked: list[RankedSignal] = []
    for signal in signals:
        score = score_signal(signal)
        ranked.append(
            RankedSignal(
                name=signal["name"].strip(),
                score=score,
                decision=classify_signal(score),
            )
        )
    return sorted(
        ranked,
        key=lambda signal: (-signal["score"], signal["name"]),
    )


def build_observatory_report(signals: Sequence[Signal]) -> str:
    """Return the complete ranked observatory report."""
    lines = ["CLOCKWORK OBSERVATORY"]
    lines.extend(
        f"{position}. {signal['name']} | {signal['score']} | {signal['decision']}"
        for position, signal in enumerate(rank_signals(signals), start=1)
```

PYTHON (CONTINUED) (CONTINUED)

```
    )
    return "\n".join(lines)
```

Use this `pyproject.toml`:

TOML

```
[project]
name = "clockwork-observatory"
version = "0.1.0"
requires-python = ">=3.10"

[tool.ruff]
line-length = 88
target-version = "py310"
src = ["src", "tests"]

[tool.ruff.lint]
select = ["E", "F", "I", "UP", "B", "RUF"]

[tool.mypy]
python_version = "3.10"
strict = true
show_error_codes = true

[tool.pytest.ini_options]
pythonpath = ["src"]
testpaths = ["tests"]
```

Use this local `.pre-commit-config.yaml`:

YAML

```
repos:
  - repo: local
    hooks:
      - id: ruff-check
        name: Ruff lint
        entry: ruff check
        language: system
        types: [python]
      - id: ruff-format-check
        name: Ruff format check
        entry: ruff format --check
        language: system
        types: [python]
      - id: mypy
        name: MyPy
        entry: mypy src
        language: system
        pass_filenames: false
      - id: pytest
        name: pytest
        entry: pytest -q
        language: system
        pass_filenames: false
        always_run: true
```

Use this `.github/workflows/quality.yml`:

YAML

```
name: Quality

on:
  pull_request:
```

YAML (CONTINUED)

```
  push:

jobs:
  quality:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-python@v5
        with:
          python-version: "3.10"
      - name: Install quality tools
        run: python -m pip install pytest ruff mypy
      - name: Lint
        run: ruff check src tests
      - name: Check formatting
        run: ruff format --check src tests
      - name: Check types
        run: mypy src
      - name: Run tests
        run: pytest -q
```

The verified reference project passes Ruff 0.15.20, MyPy 2.2.0 strict mode, pytest 9.1.1 with 11 collected cases, and the local pre-commit gate.

9. Check the observatory-specific decisions

Clockwork Observatory readiness check

Use the challenge’s source, assertions, and gate—not generic definitions—to choose.

1. Which key preserves descending score and ascending signal name?

- **A.** lambda signal: (signal["score"], signal["name"])
- **B.** lambda signal: signal["name"]
- **C.** lambda signal: (-signal["score"], -signal["name"])
- **D.** lambda signal: (-signal["score"], signal["name"])

2. What freshness penalty applies to a signal aged 80 minutes?

- **A.** 60 points
- **B.** 80 points
- **C.** 20 points
- **D.** 0 points

3. Which evidence protects caller ownership during `rank_signals`?

- **A.** A formatter pass
- **B.** Deep-copy the input before the call and assert equality afterward
- **C.** Only the exact report heading
- **D.** A MyPy success message

4. Which formatting command belongs in the observatory CI gate?

- **A.** ruff format src tests

- **B.** ruff check –fix
- **C.** ruff format –check src tests
- **D.** git add src tests

5. What is the strongest final evidence that the extraction preserved the published observatory behavior?

- **A.** The source has more functions
- **B.** Every name is long
- **C.** Ruff alone is clean
- **D.** The original progressive assertions, strict types, static checks, and exact report all pass from a clean project

Write your choices before checking the answer key at the end of this book.

10. Decide whether the challenge is complete

Evidence rubric

Evidence	Ready to record when
Behavior	Score, threshold, validation, ordering, mutation, empty, and exact-report checks pass unchanged
Readability	Extracted names expose validation, scoring, classification, ranking, and rendering responsibilities
Static quality	Ruff lint/format check and strict MyPy exit zero without broad suppression
Automation	A second all-files hook run is clean and CI repeats the check-only command contract
Reasoning	One debugging record and refactoring log connect each change to evidence
Reproducibility	The complete project works from its documented root without hidden notebook state

Record completion only when every statement is true:

- ☐ All progressive assertions pass without editing expected behavior.
- ☐ Ruff, MyPy, pytest, and format-check all exit zero.
- ☐ The second pre-commit all-files run makes no change.
- ☐ The CI workflow repeats the same four check-only commands.
- ☐ I can explain the scoring, tie, ownership, and validation boundaries.
- ☐ I retained one real failure and its controlled repair.

This button stores a self-reported marker only in this browser. It does not submit the project, grade it, verify identity, or issue a certificate.

Key points

- A maintenance rescue begins with a green behavior baseline and a named structural problem.
- Extract validation, scoring, classification, ranking, and rendering one stage at a time; rerun the smallest matching evidence.
- Types and tools should make the contract more visible, not hide problems with `Any`, ignores, or rewritten expected values.
- A local hook shortens feedback; a check-only CI workflow tests committed state on a clean runner.
- The finished artifact includes the source, configuration, exact report, debugging record, and an explanation another maintainer can follow.

Next: Documentation as Part of the Product Overview

Unit 15: Documentation as Part of the Product

Online lesson and runnable notebook

Turn a maintained Python package into documentation that real readers can follow, verify, navigate, and trust as the code changes.

- **Level:** Python Foundations · Unit 15
- **Estimated time:** 21–31 hours including the challenge
- **Unit outcome:** Design, write, test, build, and publish documentation that guides real readers from first contact to successful use, explains a stable public Python API, and remains trustworthy as the code changes.
- **Practice in:** Generated Colab notebooks for individual examples and a local project for files, Quarto builds, Git, and CI

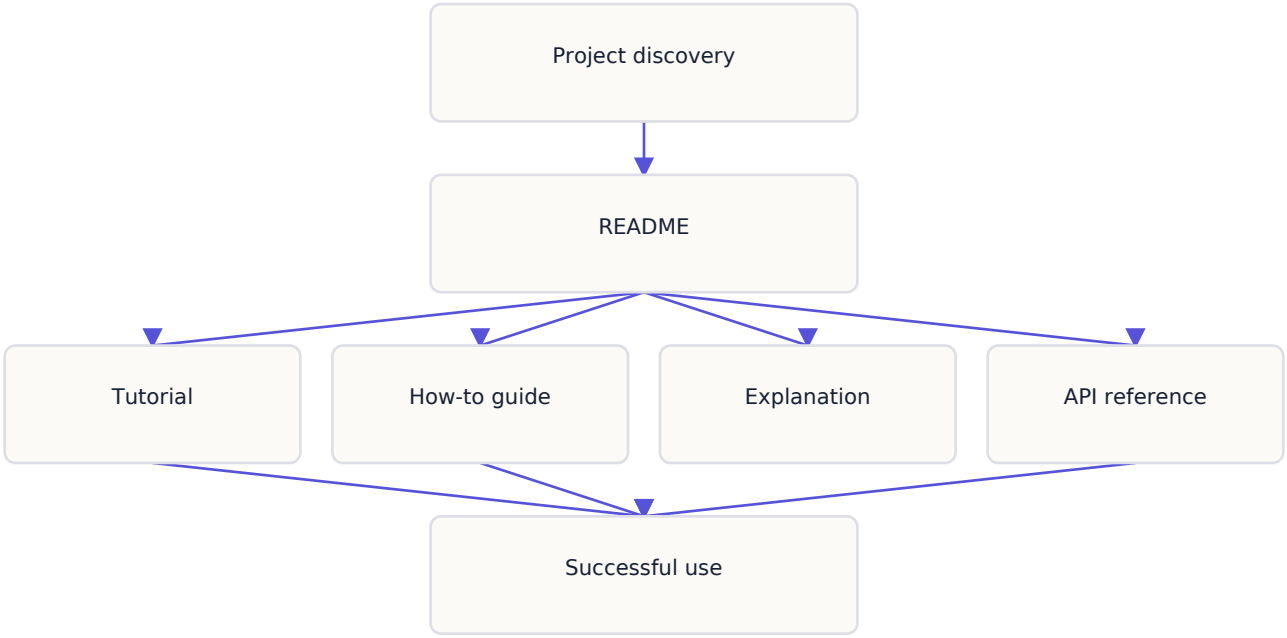
1. A working program is not yet a usable product

The Midnight Museum project reached an important state in Unit 14: its source is readable, its public functions have useful types, its tests pass, and the same quality gate runs locally and in CI. None of that tells a new person why the project exists, how to install it, which function to call, what result to expect, or how to recover from a mistake.

In this unit, you will turn that maintained code into a documented product. The finished documentation will give different readers suitable paths:

- a new visitor can discover the purpose and reach a meaningful result quickly;
- a learner can follow a guided museum-ranking tutorial;
- an existing user can change one ranking task without repeating the tutorial;
- a curious maintainer can understand why ranking is deterministic;
- a caller can look up the exact public API contract; and
- the team can rebuild and publish the site from reviewed source.

Notice that the README is an entry point, not the destination for every reader.



A beautiful page can still be wrong. A green build can still publish stale instructions. You will therefore keep two questions together throughout the unit:

1. **Does this page help its intended reader do the intended work?**
2. **What evidence shows that its commands, examples, links, and public claims still match the project?**

2. The site you will build

The lessons continue the **Midnight Museum Quest** project from Unit 14. You do not need to preserve notebook state between lessons; every page provides the files or snapshot needed for its task. The complete project gradually takes this shape:

```text
midnight-museum/
├── README.md
├── pyproject.toml
├── src/museum_quest/
│   ├── __init__.py
│   └── ranking.py
├── tests/
│   ├── test_ranking.py
│   └── test_documentation.py
├── docs/
│   ├── _quarto.yml
│   ├── index.qmd
│   ├── tutorial.qmd
│   ├── how-to-rank-exhibits.qmd
│   ├── why-ranking-is-deterministic.qmd
│   └── api.qmd
└── .github/workflows/docs.yml
```

The code remains deliberately small. Your attention should stay on the promises made to readers and on the route from source files to a verified public site.

3. What each lesson adds

Step	Lesson	What you will add
1	Design Documentation Around Reader Needs	Map reader questions, separate documentation purposes, and rebuild the README as a verified path to a first result.
2	Write Instructions People Can Follow	Turn vague procedures into clear tutorial and how-to steps with prerequisites, expected results, recovery, and accessible structure.
3	Document a Public Python API	Write caller-facing module, function, and result contracts, then inspect what `help()` and `pydoc` actually expose.
4	Keep Small Examples Honest with Doctest	Run deterministic transcript examples, diagnose their diffs, and move unsuitable cases to pytest.
5	Review Documentation Like Code	Check quick starts, examples, links, headings, rendered files, and contract changes without mistaking automation for reader feedback.
6	Build a Documentation Site with Quarto	Configure navigation and cross-references, render from a clean output directory, and inspect the generated site.
7	Publish Documentation and Keep It Trustworthy	Connect the local documentation contract to CI, deploy the verified artifact, and plan for future changes.
8	Unit Challenge	Publish a field guide that reopens the Puzzle Garden for new players.

4. Before you start

You are ready when you can:

- work from a project root and recognize `src`, `tests`, `docs`, and `pyproject.toml` from Unit 10;
- run pytest and read the earliest useful failure from Unit 13;
- run a local quality gate and read a small CI workflow from Unit 14; and
- create and follow relative file paths without relying on one notebook's hidden state.

For Lessons 1–5, you can study prose and individual Python examples in Colab. Lessons 6–7 require a local project because a documentation website is a collection of files with configuration, navigation,

output directories, and Git history. Colab remains available as a reading and scratch-work companion; it is not a substitute for a clean multi-file render.

Prepare a folder for the museum project and a short documentation log:

Reader or check	Expected result	Observed evidence	Revision
New user follows quick start	Ranked exhibit appears	Record the command and complete result	State one change you made
Doctest collects examples	Intended attempts pass	Record attempted/failed counts	Explain one repaired diff
Quarto renders site	Expected HTML pages exist	Record paths and warnings	Explain one warning decision

5. Plan realistic stopping points

The estimate assumes active work, not silent reading:

- **Reader paths and writing:** Lessons 1–2, about 5–7 hours.
- **Python-facing documentation:** Lessons 3–4, about 6–8 hours.
- **Review and site construction:** Lessons 5–6, about 6–9 hours.
- **Publication and integration:** Lesson 7 plus the challenge, about 4–7 hours.

Stop after a complete lab rather than midway through a repair sequence. When you return, rerun the nearest check before trusting the previous environment.

Use a fresh-reader test

After a page works for you, follow it again using only what the page states. Do not supply an unstated directory, import, environment variable, or remembered output. The first missing assumption is valuable documentation evidence. :::

6. What completion means here

By the end of the unit, retain:

- a README with a verified route from discovery to a meaningful result;
- tutorial, how-to, explanation, and API pages with distinct jobs;
- public docstrings and small executable examples;
- a record of one documentation failure and repair;
- a clean Quarto render with the expected navigation and pages;
- a CI/publishing explanation tied to exact commands and artifact paths; and
- the completed Puzzle Garden field guide challenge.

Progress remains local and self-reported. It is not verified assessment or certificate evidence. A Foundations final project and certificate are planned but are not available yet; they will be announced later.

Begin with Design Documentation Around Reader Needs

Design Documentation Around Reader Needs

Online lesson and runnable notebook

Map reader questions to suitable documentation and build a README whose clean quick start reaches an exact, useful result.

- **Level:** Beginner
- **Estimated time:** 2.5–3.5 hours
- **You will learn:** Choose documentation from a reader's task and verify the README path from project discovery to a meaningful Python result.
- **Practice in:** Google Colab, JupyterLab, or the Midnight Museum project

A README often becomes the first conversation between a project and a person. It cannot anticipate every question. It can tell the right reader, "This project is for you," lead them to one dependable result, and point them toward the next page for their situation.

By the end of this lesson, you should be able to answer:

1. What does this reader know before opening the project?
2. What result are they trying to achieve now?
3. Does the reader need a tutorial, a how-to guide, explanation, or reference?
4. Which promises belong in the README, and which should link elsewhere?
5. What clean evidence makes a quick start trustworthy?

1. Start with a reader and a task

"Developers" is too broad to guide a document. Compare these Midnight Museum visitors:

Reader	Current situation	Immediate question	Useful destination
Curious visitor	Has not installed the package	What does it do, and is it for me?	README opening
New user	Knows basic Python	How do I rank one exhibit successfully?	Tutorial/quick start
Returning user	Already ranked an exhibit	How do I change the ranking inputs?	How-to guide
Maintainer	Understands the API	Why are repeated rankings deterministic?	Explanation
Active caller	Is writing code now	What does `rank_exhibit` accept, return, and raise?	API reference

A good audience statement includes **context** and **work**. "For beginners" only describes presumed experience. "For a learner who has Python 3.10 and wants to rank one exhibit from a local project"

gives the author decisions to make: name the required version, identify the project root, show the import, and show the result.

Use a small record before drafting:

```python
reader = {
    "context": "Python learner at the project root",
    "goal": "rank one exhibit",
    "prerequisites": ["Python 3.10+", "project files downloaded"],
    "success": "ExhibitScore(name='Moon Dial', score=44)",
}

for label, value in reader.items():
    print(f"{label}: {value}")
```

The dictionary does not become the documentation. It forces an author to name assumptions before they disappear into prose.

Name one concrete reader

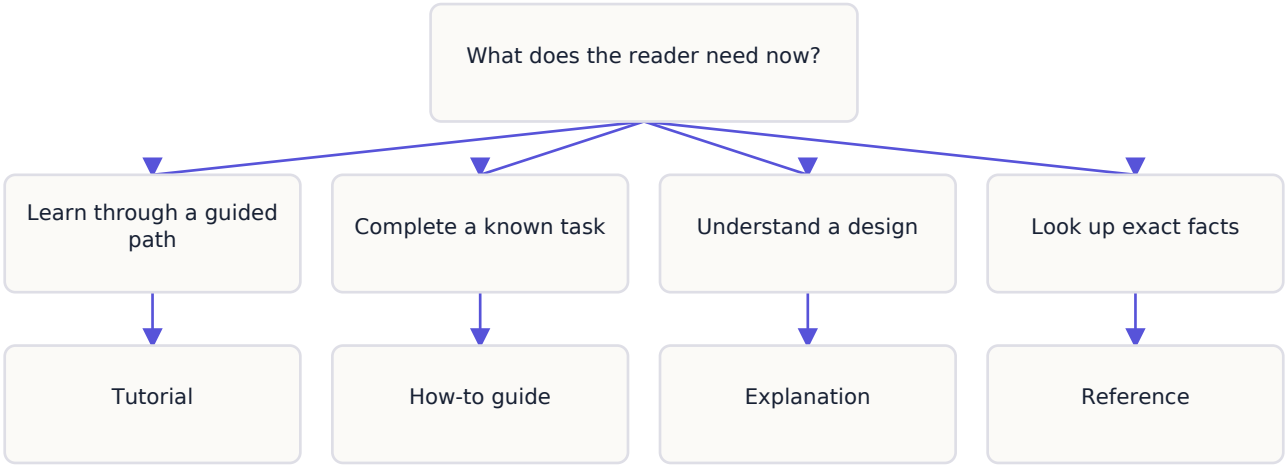
Write a reader record for someone who already completed the quick start and now needs to rank an exhibit with a different open time. Which fields change? Which page should receive that task? :::

2. Give each kind of documentation one job

Tutorials, how-to guides, explanation, and reference answer different questions. The distinction is about reader need, not file extensions.

- A **tutorial** creates a successful learning experience. It controls the path, supplies safe choices, and explains only enough to keep the learner oriented.
- A **how-to guide** helps an already-oriented reader achieve a particular result. It begins from stated prerequisites and does not stop to teach the entire system.
- **Explanation** helps a reader understand why the system behaves as it does, which alternatives exist, or which trade-off shaped a decision.
- **Reference** describes the product accurately and consistently: public signatures, parameters, returned values, exceptions, commands, and formats.

The reader’s question selects the document; the framework does not select it for them.



Consider this crowded paragraph:

Install the package, then call `rank_exhibit(name, votes, minutes_open)`. The score uses ten points per vote because visitor choices matter most. The function returns `ExhibitScore` and raises `ValueError` for negative values. To compare three exhibits, loop over their records and sort the results.

It tries to install, explain policy, state reference facts, and solve a later task at once. Separate the information:

- Quick start: install and rank one supplied exhibit.
- Explanation: why votes and time contribute different weights.
- Reference: signature, result type, and `ValueError` contract.
- How-to: rank and compare several exhibits.

Do not create empty `tutorials/`, `how-to/`, `explanation/`, and `reference/` directories merely to appear organized. Create the page that answers an actual reader question, label its purpose, and link it from the places where that question arises.

Checkpoint: choose the reader’s page

Match the question to the document

Choose the page that best fits each reader’s immediate need.

1. A learner has never used the package and wants a guided path to one successful ranking. What should they open?

- **A.** A tutorial
- **B.** A complete API index
- **C.** A design-history explanation
- **D.** A release manifest

2. A returning user knows the package and needs to rank a batch of exhibits. Which page is the best fit?

- **A.** A beginner tutorial repeated from installation

- **B.** A focused how-to guide
- **C.** An alphabetical parameter table only
- **D.** A history of scoring policy

3. A caller needs the accepted parameter types and documented exceptions for `rank_exhibit`. Where should those exact facts live?

- **A.** Only in a blog announcement
- **B.** Only in an architecture diagram
- **C.** API reference
- **D.** A contributor biography

Write your choices before checking the answer key at the end of this book.

3. Make the README a dependable doorway

A useful project README normally answers these questions in a scannable order:

1. **What is this?** Name the product and its visible purpose.
2. **Why would I use it?** Show the useful result, not internal architecture.
3. **Can I run it?** State supported Python and any important platform limit.
4. **How do I install or prepare it?** Give one maintained path and its context.
5. **What is the shortest meaningful use?** Provide a complete quick start.
6. **What should I see?** Show a small expected result.
7. **What should I read next?** Link by reader task.
8. **What are the important limits and support path?** Do not hide them below a wall of badges or history.

Here is a compact entry point:

MARKDOWN

````
# Midnight Museum Quest

Rank museum exhibits with a small, deterministic Python API.

## Requirements

Use Python 3.10 or newer. Run the commands below from the project root.

## Quick start

```python
from museum_quest import rank_exhibit

result = rank_exhibit("Moon Dial", votes=4, minutes_open=75)
print(result)
```

Expected result:

```text
ExhibitScore(name='Moon Dial', score=44)
```
````

```
MARKDOWN (CONTINUED)

Continue with the [guided museum visit](docs/tutorial.qmd "tutorial"), or see the
[public API contract](docs/api.qmd "API reference").
```

The opening tells a visitor what the package does before asking them to install anything. The requirement establishes version and working directory. The code shows every name it needs. The exact result proves more than “it works”: the reader can compare the class name, exhibit name, and score.

Keep installation honest

In a real published package, installation might be:

```
BASH
python -m pip install midnight-museum
```

In a lesson-sized source checkout that has not been published, claiming that command would be false. Use the path the learner can actually run, such as an editable install from the project root:

```
BASH
python -m pip install -e .
```

Name the context. A command that succeeds only because the author previously set `PYTHONPATH`, installed an unpublished wheel, or left a notebook import in memory is not a clean quick start.

When `pyproject.toml` declares the README, that same entry point can become package metadata:

```
TOML
[project]
name = "midnight-museum"
version = "0.1.0"
requires-python = ">=3.10"
readme = "README.md"
```

This connection increases the cost of repository-only relative links or markup that a package index cannot render. Build and inspect the distribution in Unit 10’s workflow when package publication is relevant; Unit 15 concentrates on whether the entry content remains accurate.

4. A quick start needs meaningful evidence

Syntax is only one boundary. This check proves that a text fragment parses:

```
PYTHON
quick_start = '''from museum_quest import rank_exhibit

print(rank_exhibit("Moon Dial", votes=4, minutes_open=75))
'''

compile(quick_start, "README quick start", "exec")
print("The Python syntax is valid.")
```

It does **not** prove that `museum_quest` is installed, the exported name still exists, the arguments match the public interface, or the displayed score is current.

A stronger documentation check runs the same public call and compares the promised result:

```python
from museum_quest import ExhibitScore, rank_exhibit

result = rank_exhibit("Moon Dial", votes=4, minutes_open=75)
expected = ExhibitScore(name="Moon Dial", score=44)
assert result == expected
print(result)
```

Expected output:

```text
ExhibitScore(name='Moon Dial', score=44)
```

The cleanest test executes from the environment and directory the README names. For a full project, record both context and command:

```bash
python -m venv .venv
# Activate the environment using the command for your shell.
python -m pip install -e .
python examples/quick_start.py
```

Avoid shell prompts such as `$` inside copyable command blocks. If a placeholder is required, label it:

```text
python -m pip install <path-to-project>
```

`<path-to-project>` signals replacement. `project` without brackets may look like a literal directory name.

Checkpoint: trust the quick start

Decide what the quick-start evidence proves

Choose the strongest evidence for the stated claim.

1. `compile()` accepts the README code string. What has been established?

- **A.** The package installs on every platform
- **B.** The expected score is current
- **C.** All links resolve
- **D.** The string is valid Python syntax

2. Which check gives the strongest evidence for a clean-install quick start?

- **A.** Create a fresh environment, follow only the documented steps, and compare the meaningful result
- **B.** Run the code in a notebook that already imported the package
- **C.** Read the code without executing it
- **D.** Check that the README contains a Python fence

3. Why should a quick start show a small expected result?

- **A.** It makes every test unnecessary
- **B.** It gives the reader a concrete comparison for successful behavior

- **C.** It guarantees the project has no defects
- **D.** It replaces the API reference

Write your choices before checking the answer key at the end of this book.

5. Link readers forward without trapping them

A README becomes hard to scan when it contains every advanced task, complete API detail, design discussion, contributor procedure, and release history. Keep the entry path short and use descriptive links:

MARKDOWN

```
## Learn more

- [Rank your first exhibit](docs/tutorial.qmd "tutorial")
- [Change ranking inputs](docs/how-to-rank-exhibits.qmd "ranking how-to")
- [Understand deterministic scoring](docs/why-ranking-is-deterministic.qmd "design explanation")
- [Look up the public API](docs/api.qmd "API reference")
```

“Click here” hides the destination when a reader scans links or uses assistive technology to list them. “API” is better, but “Look up the public API” also states why the reader would follow it.

Use paths relative to the file containing the link. From root `README.md`, `docs/api.qmd` points into the documentation folder. From `docs/tutorial.qmd`, `api.qmd` points to a sibling. A path beginning with a developer’s home directory can never be a portable project link.

This small standard-library check verifies known local targets:

PYTHON

```
from pathlib import Path

root = Path(".")
targets = [
    Path("docs/tutorial.qmd"),
    Path("docs/how-to-rank-exhibits.qmd"),
    Path("docs/why-ranking-is-deterministic.qmd"),
    Path("docs/api.qmd"),
]

missing = [str(target) for target in targets if not (root / target).is_file()]
assert missing == [], f"missing README targets: {missing}"
```

This is stronger than checking hand-written labels in isolation, but it is not a general Markdown link parser. Lesson 5 builds a more honest documentation review: check the source page, target, rendered navigation, and reader meaning.

6. Diagnose a README that works only for its author

Suppose a new learner reports:

TEXT

```
ModuleNotFoundError: No module named 'museum_quest'
```

The README said:

MARKDOWN

```
Run `python examples/quick_start.py` to rank an exhibit.
```

Use the evidence in order:

1. The failure happens at import, before ranking logic runs.
2. The README names neither installation nor a project-root requirement.
3. The author had previously installed the project in an existing environment.
4. Reproducing in a fresh environment confirms the missing step.
5. Add the real editable-install command and rerun the whole path.

Do not “repair” this by changing the example to import from `src` through a machine-specific path. That transfers the hidden assumption into public prose.

The README must not promise an unpublished command

If the package is not on a package index, do not tell learners to install it by its future package name. Document the source-checkout path that exists now and revise the README when distribution actually changes. :::

Checkpoint: keep the README focused

Choose what belongs at the project entrance

Use the reader journey and link evidence from this section.

1. Which README link is most useful to a returning reader who wants one specific task?

- **A.** Click here
- **B.** More
- **C.** Change ranking inputs
- **D.** docs/how-to-rank-exhibits.qmd

2. The README has grown into a 70-page API dump. What is the most reader-focused revision?

- **A.** Delete all API information without a replacement
- **B.** Add the same dump above the quick start
- **C.** Hide the quick start at the end
- **D.** Move exact API detail to navigable reference and link to it from the README

3. A quick start works only after the author manually installed the package yesterday. What should happen next?

- **A.** Document and verify the missing installation step from a fresh environment
- **B.** Describe the author’s machine as the supported environment
- **C.** Remove the import from the example
- **D.** Claim that imports are outside documentation scope

Write your choices before checking the answer key at the end of this book.

7. Lab: rebuild the museum entrance

Start with this flawed README:

````
MARKDOWN
# Tool

This uses our architecture to do rankings. Install it normally. Click
[here](file:///home/author/museum/docs/api.html) for more.

```python
print(rank_exhibit("Moon Dial", 4, 75))
```

It should work. The implementation has a scorer, validators, dataclasses,
private helpers, and other details. Version history follows...
````

Create a revised `README.md` that includes:

- a specific project name and one-sentence value statement;
- Python version and project-root context;
- the actual installation path for the project state you have;
- a complete import and call;
- the exact expected `ExhibitScore` result;
- a limitations statement and support path;
- relative, descriptive links to tutorial, how-to, explanation, and API pages;
- no private implementation inventory in the entry journey.

Then check it:

1. Create or reset a clean environment.
2. Follow only the README.
3. Record the complete command that produces the first result.
4. Compare the result with the documented value.
5. Check that each local target exists.
6. Ask which reader question each linked page answers.
7. Introduce one stale expected score, observe the mismatch, and repair the documentation or code according to the actual public contract.

Hint: choose the first visible success

Use one exhibit, one public import, and one returned value. Batch ranking, custom policies, and internal helpers belong after the entry path.

Reveal a representative README after attempting the checks

````
MARKDOWN
# Midnight Museum Quest

Rank museum exhibits with a small, deterministic Python API.
````

MARKDOWN (CONTINUED)

````
## Requirements

Use Python 3.10 or newer. Run commands from the project root.

## Installation

```text
python -m pip install -e .
```

## Quick start

```python
from museum_quest import rank_exhibit

result = rank_exhibit("Moon Dial", votes=4, minutes_open=75)
print(result)
```

Expected result: `ExhibitScore(name='Moon Dial', score=44)`.

## Limitations

This teaching package stores no rankings between runs.

## Support

Open a project issue with the command, complete error, and Python version.

## Learn more

- [Follow the guided museum visit](docs/tutorial.qmd "tutorial")
- [Change ranking inputs](docs/how-to-rank-exhibits.qmd "ranking how-to")
- [Understand deterministic scoring](docs/why-ranking-is-deterministic.qmd "design explanation")
- [Look up the public API](docs/api.qmd "API reference")
````

A different structure is valid if the path remains short, complete, and verified for the named reader.

Key points

- Begin with one reader’s context, question, and observable success.
- Tutorials, how-to guides, explanation, and reference serve different needs.
- A README should establish value and a verified first result, then route readers to deeper pages.
- Valid syntax is weaker evidence than a clean installation, public call, exact result, and resolved links.
- Document the project state that exists now, not an imagined future release.

References

- Diátaxis in five minutes
- The difference between tutorials and how-to guides
- GitHub guidance for repository README files
- Python project metadata for a README

Next: Write Instructions People Can Follow

Write Instructions People Can Follow

Online lesson and runnable notebook

Write and repair technical instructions with explicit prerequisites, focused steps, observable results, recovery guidance, and accessible structure.

- **Level:** Beginner
- **Estimated time:** 2.5–3.5 hours
- **You will learn:** Turn an ambiguous procedure into instructions another person can follow, verify, scan, and recover from without the author's hidden knowledge.
- **Practice in:** A local Markdown/QMD page, with Colab or JupyterLab for the Python results

Readers usually open instructions because they want a result, not because they want to admire prose. The writing is successful when they can tell whether the page fits their situation, perform each action safely, recognize the expected state, and recover from a predictable mistake.

This lesson asks:

1. What must a reader know or prepare before step 1?
2. Where should each command run, and which text is literal or replaceable?
3. What observable result tells the reader that a step succeeded?
4. How do headings, link text, and image descriptions affect navigation?
5. How can a fresh-reader test expose assumptions the author no longer notices?

1. Promise one result in the title and opening

Compare these titles:

- `More details`
- `Museum operation information`
- `Change the Open Time for an Exhibit Ranking`

Only the third title lets a returning user predict the page's task. The opening should confirm the result and identify the reader:

MARKDOWN

```
# Change the Open Time for an Exhibit Ranking

Use this guide after you have completed the quick start. You will change the
`minutes_open` input, rerun one public call, and verify the new score.
```

The page does not promise to explain the entire ranking formula or teach package installation. It states a prerequisite by outcome ("completed the quick start") and a visible result.

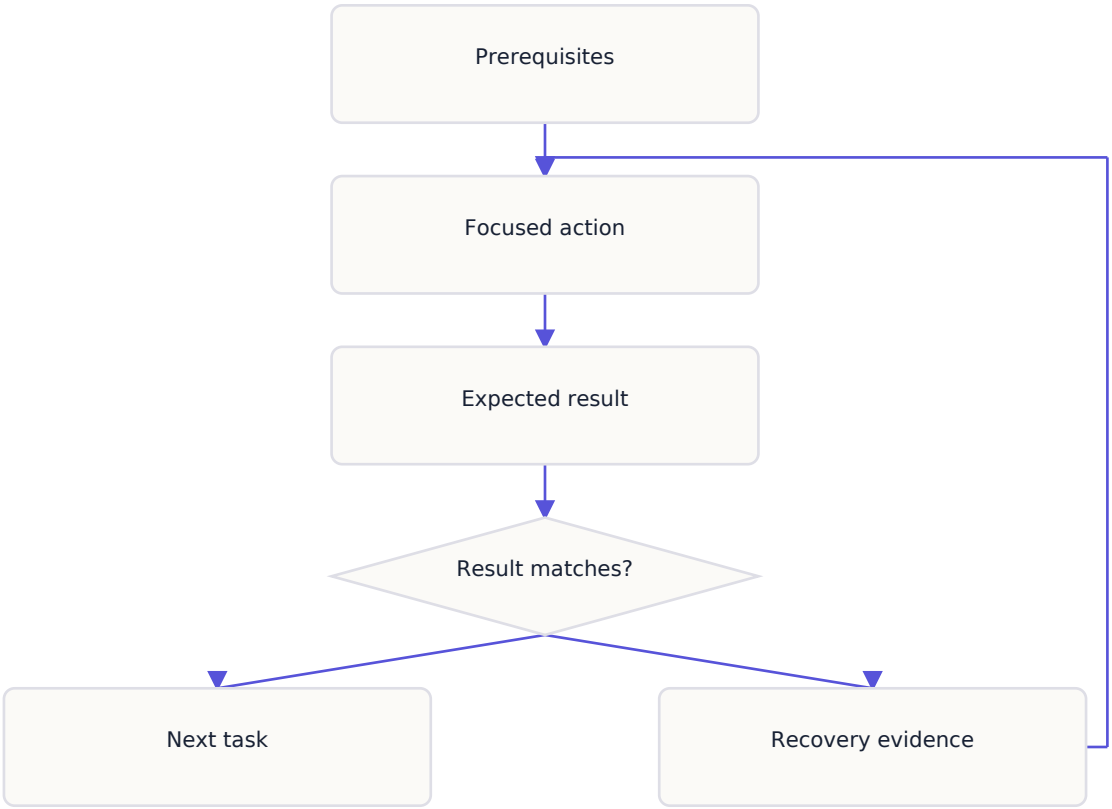
A tutorial opening makes a different promise:

```
MARKDOWN
# Rank Your First Museum Exhibit

In this guided visit, you will prepare the lesson project, rank the supplied
Moon Dial exhibit, and compare the returned score with a known result. No
ranking-policy knowledge is required.
```

Both pages contain actions, but their relationship with the reader differs. A tutorial chooses safe inputs and teaches orientation. A how-to respects an already-oriented reader’s time and concentrates on the requested change.

A usable instruction page connects preparation, action, evidence, and recovery.



Result matches? → Next task: Yes

Result matches? → Recovery evidence: No

2. Put prerequisites before dependent actions

“Install it normally” forces the reader to guess the tool, directory, package state, and supported Python version. State only prerequisites that affect this page:

```
MARKDOWN
## Before you begin

- Use Python 3.10 or newer.
- Complete the README quick start once.
- Open a terminal at the `midnight-museum/` project root.
- Keep the existing tests unchanged.
```

A prerequisite is useful when a reader can check it. “Know Python” is vague. “Be able to run `python --version` and see Python 3.10 or newer” is observable:

```bash
python --version
```

Expected shape:

```text
Python 3.13.5
```

The exact patch version may differ. Say which part matters rather than showing one output and implying it is universal.

Name the command context

These commands are not equivalent when run from arbitrary directories:

```bash
python examples/rank_moon_dial.py
pytest tests/test_ranking.py -q
quarto render docs
```

A short context line prevents three different path failures:

Run the following commands from the project root, the directory containing `pyproject.toml` and `docs/`.

Do not put a shell prompt inside a copyable block:

```text
$ python examples/rank_moon_dial.py
```

The `$` may be mistaken for part of the command. Use prose or a block label to distinguish command from output.

Checkpoint: make preparation observable

Prepare the reader before step 1

Choose the revision that removes the hidden assumption.

1. A command uses `docs/_quarto.yml`, but the page never names a working directory. Which revision helps most?

- **A.** Say that paths are obvious
- **B.** Tell the reader to run it from the project root and identify that root by a visible file
- **C.** Replace every path with the author’s home directory
- **D.** Move the command into an unexplained screenshot

2. The project needs Python 3.10 or newer. Which prerequisite is checkable?

- **A.** Use a modern Python
- **B.** Know programming
- **C.** Run `python --version` and confirm the major/minor version is at least 3.10

- **D.** Upgrade everything without reading the output

3. Which opening best fits a tutorial for a new user?

- **A.** Reference: all function signatures
- **B.** Change one setting after completing setup
- **C.** Implementation history since version 0.1
- **D.** Follow a guided path to rank the supplied exhibit; no ranking-policy knowledge is required

Write your choices before checking the answer key at the end of this book.

3. Make each step answer three questions

A procedural step should make three facts easy to find:

1 **Action:** What should the reader do?

2 **Location or input:** Where, or with which exact value?

3 **Evidence:** What changes or appears afterward?

This step hides all three:

Update it and run things. If it fails, fix the configuration.

A focused version is longer but less costly:

MARKDOWN

````
1. Open `examples/rank_moon_dial.py` from the project root.
2. Change only `minutes_open=75` to `minutes_open=105`.
3. Save the file and run:

   ```text
   python examples/rank_moon_dial.py
   ```

4. Confirm that the complete result is:

   ```text
   ExhibitScore(name='Moon Dial', score=46)
   ```
````

The controlled input change adds one complete 30-minute period. The expected score changes from 44 to 46. If the policy needs explanation, link to the design page after the reader verifies the task.

Separate literal text from placeholders

This command is ambiguous:

TEXT

```
python examples/rank.py name votes minutes
```

The words might be literal arguments or placeholders. Mark replacement points and supply a concrete invocation:

TEXT

```
python examples/rank.py <name> <votes> <minutes-open>
python examples/rank.py "Moon Dial" 4 75
```

If angle brackets have special meaning in the shell being taught, describe the notation and tell the reader not to type the brackets. Better still, prefer a real command when the task allows it.

Show expected state, not ceremonial success

“Success!” teaches nothing if the program always prints it. Prefer evidence that contains the task’s important values:

```
PYTHON
from museum_quest import rank_exhibit

result = rank_exhibit("Moon Dial", votes=4, minutes_open=105)
assert result.name == "Moon Dial"
assert result.score == 46
print(result)
```

Expected output:

```
TEXT
ExhibitScore(name='Moon Dial', score=46)
```

The assertions make the promised name and score explicit. They still check one case, not all behavior.

4. Put recovery beside the failure it repairs

A common failure belongs near the step that can cause it. Avoid a remote “Troubleshooting” dump that makes readers search for every error.

```
MARKDOWN

**If Python cannot import museum_quest**

Confirm that the virtual environment is active and that you ran
`python -m pip install -e .` from the directory containing `pyproject.toml`.
Then rerun the import command. Do not add a machine-specific `sys.path` entry.
:::
```

A recovery note should use exact evidence. “If it doesn’t work, reinstall” may erase useful state and still leave the original cause.

Consider this output:

```
TEXT
python: can't open file '/work/docs/examples/rank_moon_dial.py':
[Errno 2] No such file or directory
```

The path includes `/work/docs/examples`, but the example file is under `/work/examples`. The earliest useful hypothesis is wrong working directory, not a defect in `rank_exhibit`.

A good recovery sequence is:

1. Print or inspect the current directory.
2. Locate `pyproject.toml` and `examples/`.
3. Move to that project root.
4. Rerun the unchanged command.

5 Record the path evidence if it still fails.

Checkpoint: write actions and recovery

Make each step verifiable

Choose the instruction that best supports the reader’s task.

1. Which instruction makes one controlled edit?

- **A.** Change `minutes_open=75` to `minutes_open=105`, save, and leave every other input unchanged
- **B.** Improve all values and configuration
- **C.** Experiment until the score looks right
- **D.** Replace the function with another design

2. Which expected result is most useful after ranking the Moon Dial?

- **A.** No news is good news
- **B.** The complete `ExhibitScore(name='Moon Dial', score=46)` value
- **C.** A screenshot cropped before the result
- **D.** A message that says only `done`

3. The error path shows `/work/docs/examples/rank_moon_dial.py`, but the file is under `/work/examples`. What should the reader check first?

- **A.** Rewrite the score formula
- **B.** Delete the virtual environment
- **C.** Whether the command ran from the project root
- **D.** Change the expected score

Write your choices before checking the answer key at the end of this book.

5. Let headings and links carry meaning

Readers often scan a page before reading paragraphs. Read this outline alone:

```text
Overview
Details
More details
Notes
Other
```

It does not reveal the task. A meaningful outline does:

```text
Before you change the ranking
Change only the open-time input
Confirm the new ExhibitScore
Recover from an import failure
Choose the next museum task
```

Use heading levels as structure, not decoration. A page title is the top-level heading supplied by Quarto. Major sections use ##; subsections inside them use ###. Do not choose #### because it looks visually smaller while skipping the hierarchy a screen reader or outline tool depends on.

Write links that make sense outside the sentence

Compare:

```
MARKDOWN
For API details, [click here](api.qmd "API reference").

[Look up the `rank_exhibit` contract](api.qmd "API reference").
```

The second link remains meaningful in a list of links. Avoid repeating the raw URL as the visible label unless the URL itself is what readers need to copy.

Describe the image’s purpose

An informative diagram needs alternative text that communicates its relevant content:

```
MARKDOWN
![The README branches to a tutorial, how-to guide, explanation, and API reference.](reader-paths.svg)
```

“Diagram” or “image123” merely announces that an image exists. A decorative flourish should not force a screen-reader user through a useless filename; use the output format’s decorative-image mechanism or omit it when it contributes nothing.

Captions and surrounding prose can provide longer context. Alternative text should deliver the image’s purpose in this page, not inventory every color and pixel.

Tables need a reason

The reader/task matrix works as a table because rows share the same comparable fields. A single sequence of steps belongs in a numbered list. A table with paragraphs, code blocks, and hidden reading order can be harder on narrow screens and assistive technology.

6. Edit vague prose into domain language

Generated-sounding text often replaces concrete actions with abstractions:

Leverage the configuration capability to facilitate the production of the artifact and ensure successful execution.

Ask what the reader actually does:

In `docs/_quarto.yml`, add `api.qmd` to the sidebar. Run `quarto render docs` and confirm that `_site/api.html` exists.

The revision names the file, field, command, and evidence. It is shorter because it carries more information per sentence.

Watch for these repair opportunities:

Vague wording	Question to ask	Concrete replacement
“it”	Which file, command, or value?	“the `docs/_quarto.yml` file”

Vague wording	Question to ask	Concrete replacement
“properly”	Which result proves that?	“with zero doctest failures”
“handle”	Validate, convert, raise, log, or return?	Name the actual action
“the artifact”	Source page or rendered site?	“`docs/_site/index.html`”
“simply”	Which prerequisite is being hidden?	State the prerequisite
“obviously”	What evidence makes it visible?	Show the evidence

Direct language does not mean abrupt language. Tell the reader why a dangerous or surprising step matters, and state consequences before the action:

Removing `docs/_site` deletes generated files only; it does not delete QMD source. Confirm that your project config uses `_site` as its output directory before cleaning it.

Checkpoint: improve structure and language

Make the page scannable and accessible

Choose the revision that communicates the intended structure or purpose.

1. A diagram shows the README branching to four reader paths. Which alternative text is useful here?

- **A.** diagram.png
- **B.** A colorful picture
- **C.** Documentation
- **D.** The README links to tutorial, how-to, explanation, and API reference paths

2. Which link text remains meaningful when read by itself?

- **A.** Look up the `rank_exhibit` API contract
- **B.** Click here
- **C.** More
- **D.** This page

3. Which heading best helps a reader scan for recovery from an import error?

- **A.** Details
- **B.** Recover from `ModuleNotFoundError`
- **C.** Other
- **D.** Notes 2

Write your choices before checking the answer key at the end of this book.

7. Lab: repair a guide using only its words

The museum team wrote this page:

```
MARKDOWN
# Configuration

Make sure everything is ready. Open it and change the value. Run the command.
It will work properly. If not, fix your environment. Then click
[here](api.qmd "API reference"). The image below explains it.

![diagram](ranking.png)
```

Turn it into a how-to guide for changing `minutes_open` from 75 to 105.

Your revision must include:

- a task-specific title and one-sentence outcome;
- checkable prerequisites and a project-root statement;
- the exact filename and old/new value;
- a copyable command without a prompt character;
- the complete expected result with score 46;
- recovery for the wrong-working-directory error shown earlier;
- a descriptive API link;
- a useful image description, or a reason to remove the image;
- headings whose outline explains the task.

Follow the finished guide from a clean terminal state. Keep a table:

Step	What the page told you	What you observed	Missing assumption or revision
1			
2			
3			

Then ask a second question: if the same content were a tutorial for someone who had never ranked an exhibit, what orientation and safe supplied choices would you add? Do not bloat the how-to; describe the separate tutorial changes.

Hint: write the expected result before polishing prose

Anchor the page on `ExhibitScore(name='Moon Dial', score=46)`. Work backward to the one input edit and the context needed to produce it.

Reveal a concise solution outline

A strong guide can use this outline:

```
MARKDOWN
# Change the Open Time for an Exhibit Ranking
```

MARKDOWN (CONTINUED)

````
Use this guide after the README quick start to rerun the Moon Dial with 105 open
minutes.

## Before you change the ranking

Run from the project root with the editable installation active.

## Change only `minutes_open`

In `examples/rank_moon_dial.py`, replace `75` with `105`. Run:

```text
python examples/rank_moon_dial.py
```

## Confirm the returned score

Expect `ExhibitScore(name='Moon Dial', score=46)`.

## Recover from a missing example path

If the error path contains `docs/examples`, return to the directory containing
`pyproject.toml` and rerun the unchanged command.

## Look up related facts

[Look up the `rank_exhibit` API contract](api.qmd "API reference").
````

Your wording can differ. Verify that every action, location, and result remains observable.

Key points

- A useful instruction page promises one result for a named reader context.
- Put checkable prerequisites and working-directory context before dependent actions.
- Focus each step on an action, exact input or location, and observable result.
- Place evidence-based recovery near the failure-producing step.
- Meaningful headings, links, and image descriptions help every reader scan and navigate.
- Replace vague abstractions with the real file, command, value, result, or failure.

References

- W3C tips for writing accessible web content
- GitHub Docs writing practices
- Diátaxis how-to guide principles
- Diátaxis tutorial principles

Next: Document a Public Python API

Document a Public Python API

Online lesson and runnable notebook

Write caller-facing module, function, and result contracts, then inspect the interface that help, pydoc, signatures, and docstrings expose.

- **Level:** Beginner
- **Estimated time:** 3–4 hours
- **You will learn:** Document public Python behavior, inputs, results, side effects, and anticipated failures without duplicating signatures or narrating implementation.
- **Practice in:** Google Colab, JupyterLab, or a local `museum_quest` package

A caller can see a function's name and signature, yet still not know what its numbers mean, which inputs are valid, whether it mutates anything, or what to do with a failure. A useful public docstring completes that contract and appears where Python users naturally look: editor help, `help()`, `pydoc`, and generated reference.

This lesson asks:

1. Which names are public enough to document as promises?
2. What does a signature or type annotation already say, and what remains unexplained?
3. What belongs in module, class, method, property, and function docstrings?
4. Which exceptions and side effects matter to a caller?
5. How can you inspect the documentation Python actually exposes?

1. Document the interface a caller is invited to use

The package initializer selects its public entrance:

```python
from .ranking import ExhibitScore, rank_exhibit

__all__ = ["ExhibitScore", "rank_exhibit"]
```

A caller should not need `_score_time_bonus` or `_validate_count` to complete a normal task. Documenting every private helper as if it were stable creates more promises than the project intends to keep.

Public does not mean "every name without an underscore" by magic. Look for evidence:

- the package exports the name;
- the README or examples invite callers to use it;
- tests protect caller-visible behavior;
- another package or application depends on it; or
- maintainers explicitly commit to compatibility.

Comments, docstrings, and long-form pages serve different readers:

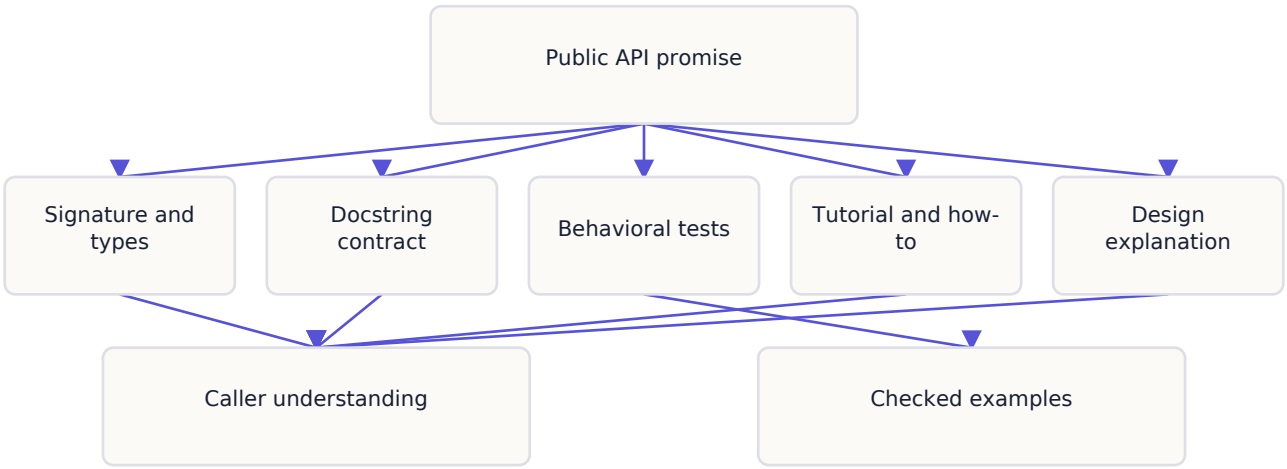
```python
POINTS_PER_VOTE = 10  # Keep the public score policy visible near the formula.


def rank_exhibit(name: str, votes: int, minutes_open: int) -> "ExhibitScore":
    """Return the score for one named exhibit."""
    ...
```

- The comment explains a source-level maintenance decision.
- The docstring states caller-visible behavior at runtime.
- A tutorial can guide a new user through a supplied call.
- An explanation page can defend the scoring policy and trade-offs.
- API reference can organize signatures and contracts for lookup.

Do not paste the same paragraph into all five places. Decide where each fact has one maintained source and link when a reader needs more context.

The signature, annotation, docstring, tests, and longer pages contribute different evidence about one public promise.



Checkpoint: choose the documentation layer

Put each fact where readers look for it

Choose the documentation layer that best fits the fact.

1. The integer parameter `minutes_open` is measured in minutes, not seconds. Where should a caller see that contract?

- **A.** Only in a private helper comment
- **B.** Only in Git history
- **C.** The public parameter documentation
- **D.** Nowhere because the type is `int`

2. A maintainer wants to understand why votes count more than open time. Which page best develops that reasoning?

- **A.** A list of private names
- **B.** The return annotation
- **C.** A one-line installation command
- **D.** A scoring-policy explanation

3. Which name should normally be omitted from the stable caller reference?

- **A.** The private helper `_score_time_bonus`
- **B.** The exported `rank_exhibit` function
- **C.** The exported `ExhibitScore` result
- **D.** The documented `ValueError` boundary

Write your choices before checking the answer key at the end of this book.

2. Let the summary line state behavior

PEP 257 recommends a short summary that fits on one line. For a function, begin with the action or returned meaning:

```python
def normalize_label(value: str) -> str:
    """Return a display label with outer whitespace removed."""
    return value.strip()
```

Avoid repeating the signature:

```python
def normalize_label(value: str) -> str:
    """normalize_label(value: str) -> str"""
    return value.strip()
```

Python and editor tools already inspect the signature. The useful missing fact is what “normalize” means here. It removes outer whitespace; it does not change capitalization, Unicode normalization, or internal spaces.

A multi-line docstring keeps the summary separate:

```python
def score_time_bonus(minutes_open: int) -> int:
    """Return two points for each complete 30-minute period.

    Partial periods do not contribute points. ``minutes_open`` must be
    non-negative and is measured in minutes.
    """
    if minutes_open < 0:
        raise ValueError("minutes_open must be non-negative")
    return minutes_open // 30 * 2
```

The summary supports indexes and compact help. The body names rounding, validation, and units—facts the implementation alone should not force every caller to reverse engineer.

Types complement domain meaning

Compare what this signature communicates:

```python
def rank_exhibit(
    name: str,
    votes: int,
    minutes_open: int,
) -> ExhibitScore:
    ...
```

It says:

- the caller supplies text and two integers;
- the function returns an `ExhibitScore`;
- argument order and names are visible.

It does not say:

- whether empty names are accepted;
- whether counts can be negative;
- whether open time is seconds or minutes;
- how partial periods behave;
- whether the function writes a file;
- which exception communicates invalid data.

Do not duplicate `votes: int` as “votes (int): an integer.” Add the domain role: “Non-negative visitor vote count.”

3. Write the complete caller contract

The museum function uses a consistent Google-style layout, but the information matters more than the punctuation:

```python
from dataclasses import dataclass


@dataclass(frozen=True)
class ExhibitScore:
    """Store the public result of ranking one exhibit.

    Attributes:
        name: Visitor-facing exhibit name.
        score: Total ranking points.
    """

    name: str
    score: int


def rank_exhibit(name: str, votes: int, minutes_open: int) -> ExhibitScore:
    """Return the score for one named exhibit.

    Each vote contributes 10 points. Each complete 30-minute period contributes
    2 points; a partial period contributes no time points.
```

PYTHON (CONTINUED)

```
    Args:
        name: Non-empty visitor-facing exhibit name.
        votes: Non-negative visitor vote count.
        minutes_open: Non-negative open duration measured in minutes.

    Returns:
        The exhibit name and calculated score as an immutable result.

    Raises:
        ValueError: If the name is blank or either count is negative.

    Examples:
        >>> rank_exhibit("Moon Dial", votes=4, minutes_open=75)
        ExhibitScore(name='Moon Dial', score=44)
    """
    clean_name = name.strip()
    if not clean_name:
        raise ValueError("name cannot be blank")
    if votes < 0 or minutes_open < 0:
        raise ValueError("votes and minutes_open must be non-negative")
    score = votes * 10 + minutes_open // 30 * 2
    return ExhibitScore(name=clean_name, score=score)
```

Read the contract from the caller's side:

- **Behavior:** one exhibit becomes one result; it does not save or print.
- **Policy:** votes and complete time periods contribute named amounts.
- **Inputs:** names are cleaned; values have units and non-negative boundaries.
- **Result:** an immutable value carries name and score.
- **Failures:** anticipated invalid inputs use `ValueError` with two boundaries.
- **Example:** a small deterministic public call illustrates the ordinary path.

Do not list exceptions the function does not deliberately expose. Hardware failure, `MemoryError`, or a bug in a future dependency is not an anticipated caller contract merely because it is theoretically possible.

Side effects belong in the promise

A function that writes a file must say so:

PYTHON

```
from pathlib import Path


def save_score(result: ExhibitScore, destination: Path) -> None:
    """Write one exhibit score as UTF-8 text, replacing `destination`.

    Args:
        result: Public result to serialize.
        destination: File to create or replace.

    Raises:
        OSError: If the destination cannot be written.
    """
    destination.write_text(
        f"{result.name}: {result.score}\n",
        encoding="utf-8",
    )
```

`-> None` does not communicate replacement, encoding, newline shape, or the anticipated `OSError`. A caller deciding whether the function is safe needs those facts.

Checkpoint: complete the public contract

Document facts types cannot carry

Choose the contract statement supported by the implementation.

1. What should the `minutes_open` documentation say about 75 minutes?

- **A.** All 75 minutes become separate bonus periods
- **B.** Only two complete 30-minute periods contribute time points
- **C.** Open time is interpreted as seconds
- **D.** Partial periods raise an exception

2. What important fact is missing from `minutes_open: int`?

- **A.** That Python has integers
- **B.** The function name
- **C.** The minutes unit and non-negative boundary
- **D.** The source filename

3. Which caller-visible fact belongs in `save_score` documentation?

- **A.** The author’s editor theme
- **B.** The private variable name
- **C.** Every possible operating-system implementation detail
- **D.** The destination is replaced using UTF-8 text

Write your choices before checking the answer key at the end of this book.

4. Document modules, classes, methods, and properties at their level

A module docstring tells a reader what the module groups and which public entrances matter:

```python
"""Rank museum exhibits using explicit votes and open-time inputs.

Use `rank_exhibit` for the public calculation and `ExhibitScore` for its
immutable result. This module performs no file or network I/O.
"""
```

A class docstring explains what an instance represents and important invariants. It should not copy every method’s documentation.

```python
class ExhibitCatalog:
    """Collect unique exhibits by their visitor-facing name."""

    def add(self, result: ExhibitScore) -> None:
```

PYTHON (CONTINUED)

```
        """Add `result`, raising `ValueError` when its name already exists."""
        ...

    @property
    def count(self) -> int:
        """Return the number of stored exhibits."""
        ...
```

The method states its change and duplicate-name failure. The property states the meaning of the exposed number. A constructor docstring is useful when creating the object has rules that the class summary does not explain.

Recognize styles without turning them into rival contracts

Projects commonly encode parameters and results with Google, NumPy, or reStructuredText/Sphinx conventions. For example:

PYTHON

```
def rank_exhibit(name: str, votes: int, minutes_open: int) -> ExhibitScore:
    """Return the score for one named exhibit.

    Parameters
    ----------
    name
        Non-empty visitor-facing exhibit name.
    votes
        Non-negative visitor vote count.
    minutes_open
        Non-negative open duration in minutes.

    Returns
    -------
    ExhibitScore
        Immutable exhibit name and calculated score.
    """
    ...
```

The syntax differs from `Args:` and `Returns:`, but the caller needs the same facts. Choose a house style compatible with the project's tooling, use it consistently, and review information quality rather than debating section names in isolation.

5. Inspect what Python exposes

A source docstring is not useful merely because it exists. Inspect the caller's view:

PYTHON

```
import inspect

print(inspect.signature(rank_exhibit))
print(rank_exhibit.__doc__.splitlines()[0])
```

Expected output begins:

TEXT

```
(name: str, votes: int, minutes_open: int) -> __main__.ExhibitScore
Return the score for one named exhibit.
```

The exact qualified return name can differ by module context. The important checks are parameter order/names and the summary line.

Use built-in help:

```
PYTHON
help(rank_exhibit)
```

From a local installed project, `pydoc` produces text outside the interpreter:

```
BASH
python -m pydoc museum_quest.rank_exhibit
```

Or inspect the module:

```
BASH
python -m pydoc museum_quest
```

`pydoc` imports the module. If importing starts an interactive prompt, writes a file, reads secrets, or launches the application, documentation generation will trigger those side effects. Unit 10's `if __name__ == "__main__":` boundary and Unit 14's import-safe design matter here.

Diagnose a stale promise

Suppose help says:

```
TEXT
Raises:
    ValueError: If votes is negative.
```

But the implementation now also rejects blank names and negative open time. Passing tests do not repair prose automatically. Choose the source of truth from an explicit public decision:

1. Confirm intended behavior in tests and release expectations.
2. Reproduce each boundary.
3. Update the docstring to match the intended contract.
4. Update longer reference or how-to recovery that mentions the old behavior.
5. Rerun help, doctest, pytest, and the documentation build.

Do not change code to accept invalid values merely to preserve stale prose, and do not change prose to excuse an accidental regression. Resolve the contract first.

Checkpoint: inspect and repair exposed documentation

Read the interface a caller receives

Choose the action supported by the displayed evidence.

1. Which tool shows the runtime signature and docstring through Python's built-in help system?

- **A.** `help(rank_exhibit)`
- **B.** A screenshot of the source tree

- **C.** The Git branch name
- **D.** A formatter check

2. Why can automatic API documentation expose an import-design problem?

- **A.** It converts every function to a class
- **B.** Tools may import modules to inspect their objects and docstrings
- **C.** It disables all type annotations
- **D.** It ignores public names

3. The implementation intentionally rejects blank names, but the public docstring omits that boundary. What should the team do?

- **A.** Ignore the mismatch because tests pass
- **B.** Delete validation
- **C.** Confirm the intended contract, update every affected document, and rerun the checks
- **D.** Add a private comment only

Write your choices before checking the answer key at the end of this book.

6. Repair docstrings that create false confidence

Implementation narration

PYTHON
```
def score_time_bonus(minutes_open: int) -> int:
    """Divide minutes_open by 30 with // and multiply the result by 2."""
    return minutes_open // 30 * 2
```

The sentence will become stale if implementation changes, and it makes the caller reconstruct the policy. Prefer: “Return two points for each complete 30-minute period.”

Impossible certainty

PYTHON
```
def load_catalog(path: str) -> list[str]:
    """Always load a valid catalog without errors."""
    ...
```

File paths can be missing, unreadable, or contain invalid data. Document anticipated failure and validation instead of promising away the boundary.

Copying types without meaning

PYTHON
```
def rank_exhibit(name: str, votes: int, minutes_open: int) -> ExhibitScore:
    """Rank an exhibit.

    Args:
        name: A string.
        votes: An integer.
        minutes_open: An integer.
    """
    ...
```

Add non-empty visitor-facing name, non-negative vote count, and non-negative minutes. Those facts help a caller choose values.

Leaking a private mechanism

PYTHON

```
def rank_exhibit(name: str, votes: int, minutes_open: int) -> ExhibitScore:
    """Call `_validate_count`, then `_score_time_bonus`, then create `_Result`."""
    ...
```

Private helpers can change during a behavior-preserving refactor. The public contract should survive that change.

7. Lab: document and inspect a small public module

Create `src/museum_quest/ranking.py` with:

- a module docstring;
- frozen `ExhibitScore` with meaningful attribute documentation;
- the complete `rank_exhibit` signature and implementation from this lesson;
- one private scoring helper whose name does not enter public reference.

Create `src/museum_quest/__init__.py` that exports only the result and public function. Then:

1. Predict the signature and summary line `help()` will display.
2. Inspect `inspect.signature(rank_exhibit)` and `rank_exhibit.__doc__`.
3. Run `python -m pydoc museum_quest` from the installed project.
4. Confirm public names appear and the private helper is not presented as the supported entry path.
5. Call the ordinary case and both failure categories.
6. Temporarily make the docstring promise that 75 minutes creates three complete periods. Identify the behavior/doc mismatch without editing the test to hide it.
7. Repair the promise and list every longer page that should be reviewed.

Hint: write the caller table before the prose

Use columns for name, representation/type, domain meaning, ordinary example, boundary, result, side effect, and anticipated exception. Omit facts that do not apply; translate the table into a concise contract.

Reveal a public-interface check

PYTHON

```
import inspect

from museum_quest import ExhibitScore, rank_exhibit

assert list(inspect.signature(rank_exhibit).parameters) == [
    "name",
    "votes",
    "minutes_open",
]
assert rank_exhibit.__doc__ is not None
assert rank_exhibit.__doc__.splitlines()[0] == (
```

PYTHON (CONTINUED)

```
    "Return the score for one named exhibit."
)
assert rank_exhibit("Moon Dial", 4, 75) == ExhibitScore("Moon Dial", 44)

try:
    rank_exhibit("   ", 4, 75)
except ValueError as error:
    assert str(error) == "name cannot be blank"
else:
    raise AssertionError("blank name was accepted")
```

Also inspect the actual `help()`/`pydoc` output; assertions over one summary line cannot judge the whole contract's usefulness.

Key points

- Document the interface callers are invited to depend on, not every private helper.
- Signatures and types carry structure; docstrings add domain meaning, units, boundaries, side effects, and anticipated failures.
- Summary lines state behavior rather than copying signatures or code steps.
- Module, class, method, property, and function docstrings describe their own level of the public interface.
- `help()`, `inspect`, and `pydoc` reveal the runtime documentation callers see.
- Resolve stale behavior/documentation disagreements through an explicit public contract, then update every affected source and check.

References

- PEP 257 — Docstring Conventions
- Python `pydoc` documentation
- Python `inspect` documentation
- Sphinx autodoc and its import boundary

Next: Keep Small Examples Honest with Doctest

Keep Small Examples Honest with Doctest

Online lesson and runnable notebook

Run and diagnose deterministic transcript examples with doctest, apply comparison options carefully, and move complex behavior to clearer pytest tests.

- **Level:** Beginner
- **Estimated time:** 3–4 hours
- **You will learn:** Make small public examples executable, read their exact diffs, control unstable text honestly, and recognize when pytest communicates the contract better.
- **Practice in:** A local module and terminal, with Colab or JupyterLab for isolated experiments

An example makes a promise twice: the code should be valid for the public interface, and the displayed result should match what a reader will observe. Doctest can turn a small Python transcript into an executable comparison. It is valuable when the transcript is already good documentation; it is not a reason to squeeze every behavior into a docstring.

This lesson asks:

1. How does doctest distinguish input, continuation, output, and prose?
2. Which command proves that examples were discovered rather than silently skipped?
3. How do you read a doctest diff without immediately changing expected output?
4. When do ellipsis or whitespace options express a real tolerance, and when do they hide a broken promise?
5. Which examples belong in pytest instead?

1. A transcript has exact roles

Add a small deterministic example to the public function:

PYTHON

```
from dataclasses import dataclass


@dataclass(frozen=True)
class ExhibitScore:
    """Store one exhibit ranking result."""

    name: str
    score: int


def rank_exhibit(name: str, votes: int, minutes_open: int) -> ExhibitScore:
    """Return the score for one named exhibit.

    >>> rank_exhibit("Moon Dial", votes=4, minutes_open=75)
    ExhibitScore(name='Moon Dial', score=44)
    >>> rank_exhibit("Sun Clock", votes=1, minutes_open=29)
    ExhibitScore(name='Sun Clock', score=10)
    """
```

PYTHON (CONTINUED)

```
    if votes < 0 or minutes_open < 0:
        raise ValueError("votes and minutes_open must be non-negative")
    return ExhibitScore(name, votes * 10 + minutes_open // 30 * 2)
```

Read each line by role:

- `>>>` begins an input statement as it would appear in the interactive interpreter.
- `...` continues an input statement; it is not decorative indentation.
- Lines without a prompt after an input are expected text output or representation.
- Prose outside the transcript remains documentation and is not compared as output.

Doctest uses `repr()` for a value left as an expression. That is why strings have quotes and the dataclass shows its constructor-like representation. `print()` would compare the displayed string instead.

A multi-line input uses the continuation prompt:

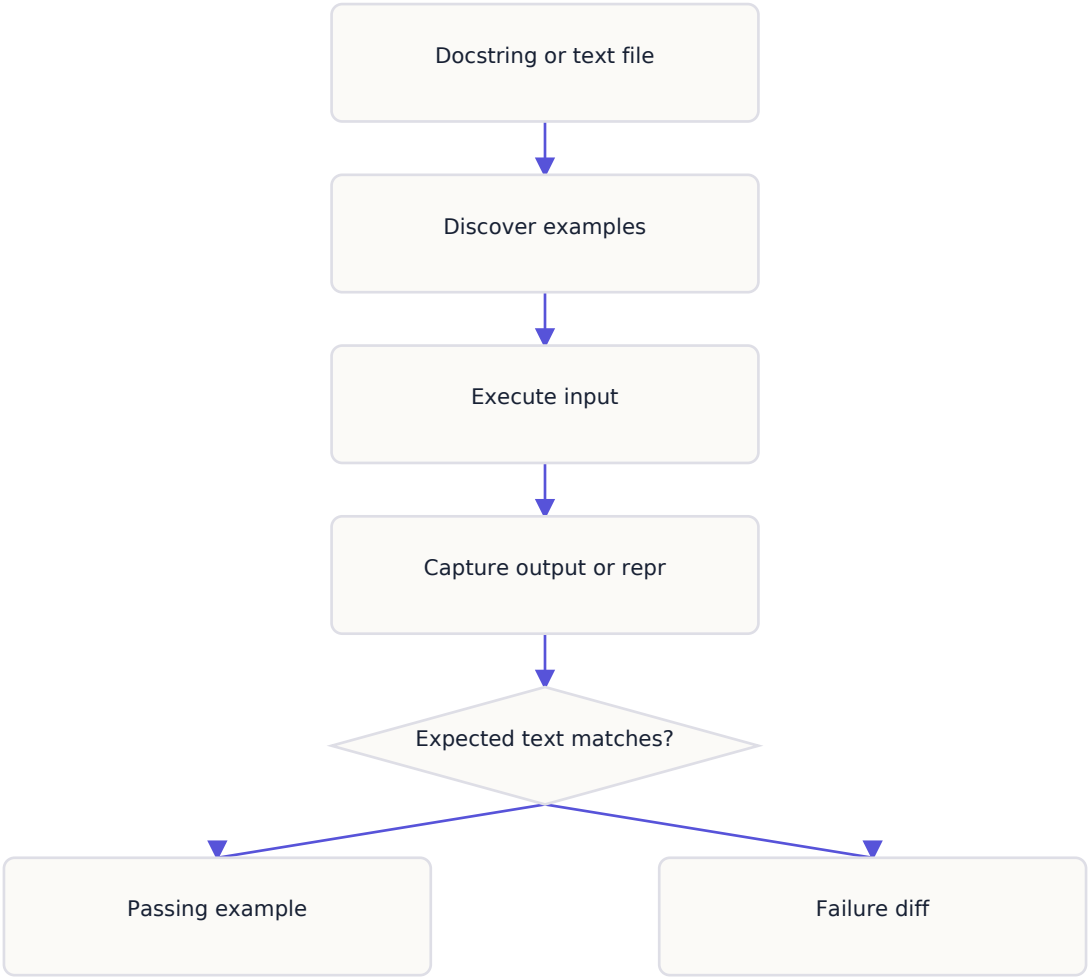
PYTHON

```
def ranked_names(records: list[ExhibitScore]) -> list[str]:
    """Return names ordered by descending score, then by name.

    >>> records = [
    ...     ExhibitScore("Moon Dial", 44),
    ...     ExhibitScore("Sun Clock", 10),
    ... ]
    >>> ranked_names(records)
    ['Moon Dial', 'Sun Clock']
    """
    ordered = sorted(records, key=lambda item: (-item.score, item.name))
    return [item.name for item in ordered]
```

The continuation indentation must be valid Python after the prompt is removed. Keep setup short enough that the reader can still see the public behavior.

Doctest discovers a transcript, executes its input, renders the observed value, and compares text.



Expected text matches? → Passing example: Yes

Expected text matches? → Failure diff: No

2. Prove that the examples were collected

From the project root, run a module directly:

```bash
PYTHONPATH=src python -m doctest -v src/museum_quest/ranking.py
```

A successful summary for the two `rank_exhibit` examples includes:

```text
2 tests in ranking.rank_exhibit
2 tests in 8 items.
2 passed and 0 failed.
Test passed.
```

Exact item counts can change as the module gains documented objects. The important evidence is the intended number of tests and zero failures.

Without `-v`, a successful run can produce no output. Silence is only useful if you already know collection occurred. Use verbose output while authoring or assert the result through the library API:

```python
import doctest
import museum_quest.ranking

result = doctest.testmod(museum_quest.ranking, verbose=False)
assert result.attempted == 2
assert result.failed == 0
```

Checking both fields matters. `failed == 0` also describes a module with zero discovered examples. The attempted count protects against a misspelled prompt, wrong module, or uncollected file.

Pytest can collect docstrings along with ordinary tests:

```bash
pytest --doctest-modules src tests -q
```

Or commit the policy in the sample project's configuration:

```toml
[tool.pytest.ini_options]
pythonpath = ["src"]
addopts = "--doctest-modules"
```

Do not enable collection and assume every prose-looking prompt is suitable. Run and review the suite, then keep the configuration versioned with the project.

Checkpoint: collect real attempts

Confirm that executable examples actually ran

Choose the evidence that supports the stated collection claim.

1. A non-verbose doctest command prints nothing. What can you conclude from that output alone?

- **A.** Exactly two examples passed
- **B.** Every docstring was discovered
- **C.** The package is fully correct
- **D.** No failure was reported, but the attempted count is still unknown

2. Which library-level check proves that two intended examples ran successfully?

- **A.** `result.attempted == 2 and result.failed == 0`
- **B.** `result.failed == 0` only
- **C.** `result is None`
- **D.** The docstring contains two `>>>` strings

3. What does pytest need before it checks examples in Python docstrings?

- **A.** A screenshot of the examples
- **B.** Doctest collection such as `--doctest-modules`
- **C.** A changed expected value
- **D.** A custom exception for every function

Write your choices before checking the answer key at the end of this book.

3. Read the diff before repairing anything

Change the expected Moon Dial score to 46 while leaving the implementation and inputs unchanged. A failure resembles:

```text
Failed example:
    rank_exhibit("Moon Dial", votes=4, minutes_open=75)
Expected:
    ExhibitScore(name='Moon Dial', score=46)
Got:
    ExhibitScore(name='Moon Dial', score=44)
```

Use all four pieces:

1. **Input:** votes 4 and open time 75.
2. **Expected:** score 46.
3. **Got:** score 44.
4. **Policy:** four votes add 40; two complete 30-minute periods add 4.

The observed 44 matches the documented policy. Repair the stale expected output to 44. If product requirements actually changed to count partial periods, then change the contract, implementation, normal tests, doctest, explanation, and release communication together. Do not pick whichever edit makes one command green fastest.

Exceptions are observable results

Document an anticipated invalid input with the exception’s final line:

```python
def rank_exhibit(name: str, votes: int, minutes_open: int) -> ExhibitScore:
    """Return the score for one named exhibit.

    >>> rank_exhibit("Moon Dial", votes=-1, minutes_open=75)
    Traceback (most recent call last):
    ...
    ValueError: votes and minutes_open must be non-negative
    """
    if votes < 0 or minutes_open < 0:
        raise ValueError("votes and minutes_open must be non-negative")
    return ExhibitScore(name, votes * 10 + minutes_open // 30 * 2)
```

Doctest treats the traceback header and final exception detail specially. The middle `...` stands for stack frames. The exception type and message remain reader-visible contract. If only incidental detail varies, use the narrowest supported comparison rather than replacing the entire expected failure with an ellipsis.

Blank output needs an explicit marker

A literal blank line ends expected output. When a printed value includes a blank line, doctest uses `<BLANKLINE>`:

```python
def score_card(result: ExhibitScore) -> str:
    """Return a two-part score card.

    >>> print(score_card(ExhibitScore("Moon Dial", 44)))
    Moon Dial
    <BLANKLINE>
    Score: 44
    """
    return f"{result.name}\n\nScore: {result.score}"
```

Use this only when the blank line helps the example. Large formatting snapshots are often clearer in an ordinary test with a named expected string.

4. Use comparison options as precise contracts

Doctest normally compares text exactly. Options can describe harmless variation, but they can also make an example meaningless.

Normalize layout when layout is not the promise

```python
def score_words() -> str:
    """Return the score column headings.

    >>> print(score_words())  # doctest: +NORMALIZE_WHITESPACE
    Exhibit    Votes    Score
    """
    return "Exhibit\tVotes\tScore"
```

`NORMALIZE_WHITESPACE` accepts equivalent runs of whitespace. This is reasonable when the example teaches the three headings, not exact tab layout. It would be wrong for a fixed-width export whose spacing is the public format.

Ellipsis should cover one unstable fragment

```python
def describe_session(session_id: str) -> str:
    """Return a short session description.

    >>> describe_session("museum-42")  # doctest: +ELLIPSIS
    'session museum-42: ... ready'
    """
    return f"session {session_id}: temporary workspace ready"
```

The stable session ID and final state remain checked. This is too broad:

```text
'...'
```

It accepts almost any non-empty representation and teaches the reader nothing. Prefer redesigning the example so unstable timestamps, paths, addresses, random values, or unordered output are not central.

Sort unordered results for the example

Instead of tolerating arbitrary set order:

```python
def exhibit_tags() -> set[str]:
    """Return the available exhibit tags.

    >>> sorted(exhibit_tags())
    ['astronomy', 'clockwork', 'light']
    """
    return {"light", "clockwork", "astronomy"}
```

The public meaning is membership, so sorting only the displayed example creates a stable teaching view without changing the returned type.

Checkpoint: choose a truthful comparison

Control textual variation without hiding behavior

Choose the comparison that preserves the important promise.

1. A function returns a set of three tags, and order is not part of its contract. How should the example become deterministic?

- **A.** Copy whichever set representation appeared today
- **B.** Use ellipsis for the entire output
- **C.** Display `sorted(exhibit_tags())` with the three exact names
- **D.** Remove expected output

2. Which ellipsis expectation is too broad to teach useful behavior?

- **A.** `'session museum-42: ... ready'`
- **B.** A traceback frame represented by `...`
- **C.** A stable prefix with one generated identifier omitted
- **D.** `'...'` for the entire returned string

3. When is `NORMALIZE_WHITESPACE` appropriate?

- **A.** When headings matter but runs of tabs/spaces do not
- **B.** When exact fixed-width layout is the published format
- **C.** To ignore different words
- **D.** To hide a missing line

Write your choices before checking the answer key at the end of this book.

5. Use a text file when the narrative is larger than one object

A tutorial-like transcript can live in `docs/examples.txt`:

```text
Rank two exhibits and compare their public results.

    >>> from museum_quest import rank_exhibit
    >>> moon = rank_exhibit("Moon Dial", 4, 75)
    >>> sun = rank_exhibit("Sun Clock", 1, 29)
    >>> moon.score > sun.score
```

TEXT (CONTINUED)

```
    True
```

Run it directly:

BASH

```
PYTHONPATH=src python -m doctest -v docs/examples.txt
```

A text file can connect several public objects without making one function’s docstring carry a long tutorial. It still needs small, deterministic setup and clear prose.

Do not confuse the source with the published QMD site automatically. If Quarto renders a different code block than the tested text file, those are two copies that can drift. Lesson 5 asks which source owns the example and how the rendered page receives it.

6. Choose pytest when structure communicates better

Doctest is a poor fit when an example needs:

- a temporary directory or several files;
- complex setup/cleanup;
- repeated boundary inputs;
- approximate numeric comparisons;
- structured assertions over many fields;
- test doubles or injected dependencies;
- intentionally unstable output; or
- a failure message clearer than a long textual diff.

This boundary table belongs in pytest:

PYTHON

```
import pytest

from museum_quest import ExhibitScore, rank_exhibit


@pytest.mark.parametrize(
    ("minutes_open", "expected_score"),
    [(0, 40), (29, 40), (30, 42), (59, 42), (60, 44)],
)
def test_complete_period_boundaries(
    minutes_open: int,
    expected_score: int,
) -> None:
    assert rank_exhibit("Moon Dial", 4, minutes_open) == ExhibitScore(
        "Moon Dial",
        expected_score,
    )
```

A docstring can keep one ordinary example and perhaps one important failure. The parametrized test protects the detailed period boundaries without overwhelming a reader who only needs the public shape.

Avoid hidden state between examples

This transcript depends on earlier mutation:

```python
def add_vote(scores: list[int]) -> int:
    """Append one vote and return the count.

    >>> values = []
    >>> add_vote(values)
    1
    >>> add_vote(values)
    2
    """
    scores.append(1)
    return len(scores)
```

The sequence is valid as one doctest, but copying the second call alone produces a different result. If independence matters to understanding, repeat the setup or use a function example that creates no hidden evolving state.

Notebook state creates a similar risk. Restart and run from the top. A locally passing cell after an old import does not prove the committed module or docstring works.

7. Diagnose discovery, comparison, and design failures separately

Use the earliest category supported by evidence:

Evidence	Likely category	Next action
`0 tests in 7 items`	Discovery	Inspect prompts, module path, and collection command
`NameError` on a public name	Setup/import	Check installation, imports, and example context
Expected/Got differ in one value	Contract or stale output	Trace inputs and intended behavior before editing
Same facts, different spacing	Text representation	Decide whether layout is public; use a narrow option if not
Timestamp/address changes	Nondeterminism	Remove unstable value or expose a stable public view
Ten setup lines obscure one assertion	Wrong test form	Move structured case to pytest

Checkpoint: decide between doctest and pytest

Choose the clearest executable evidence

Choose doctest or pytest from the reader value and setup cost.

1. Which case is a strong doctest candidate?

- **A.** Creating six temporary files and mocking a network client

- **B.** One deterministic public call whose exact returned representation teaches usage
- **C.** Fifty numeric boundary combinations
- **D.** A performance benchmark

2. Where should five complete-period boundary cases normally live?

- **A.** In five copied README sections
- **B.** Inside an all-ellipsis transcript
- **C.** A parametrized pytest test
- **D.** Only in a private comment

3. The summary reports zero failed and zero attempted. What should you investigate?

- **A.** The score formula only
- **B.** The expected output only
- **C.** Whitespace normalization only
- **D.** Discovery and collection before claiming success

Write your choices before checking the answer key at the end of this book.

8. Lab: run a mixed doctest clinic

Create `src/museum_quest/examples.py` containing six documented objects:

1. a stable `rank_exhibit` ordinary example with stale score 46;
2. a negative-votes example with the wrong exception message;
3. a tag-set example that assumes set order;
4. a session description whose expectation is only `'...'`;
5. a score-card example missing `<BLANKLINE>`; and
6. a five-row time-boundary transcript that would be clearer in pytest.

Work one failure at a time:

```bash
PYTHONPATH=src python -m doctest -v src/museum_quest/examples.py
```

For each failure, record:

Object	Attempted input	Expected	Got	Contract decision	Smallest repair

Then:

- assert the intended attempted count and zero failures with `testmod`;
- move the boundary table to a parametrized pytest test;
- keep one readable ordinary case and one useful failure in public docs;
- run pytest with doctest collection from a clean process;

- explain why each option directive is narrower than changing everything to ellipsis.

Hint: classify before changing text

Mark each issue as discovery, setup, intended-behavior mismatch, unstable representation, or unsuitable test structure. The category narrows the repair.

Reveal representative stable examples

```python
def exhibit_tags() -> set[str]:
    """Return the available exhibit tags.

    >>> sorted(exhibit_tags())
    ['astronomy', 'clockwork', 'light']
    """
    return {"light", "clockwork", "astronomy"}


def score_card(result: ExhibitScore) -> str:
    """Return a two-part score card.

    >>> print(score_card(ExhibitScore("Moon Dial", 44)))
    Moon Dial
    <BLANKLINE>
    Score: 44
    """
    return f"{result.name}\n\nScore: {result.score}"
```

The time-boundary cases remain more readable in the parametrized pytest example from section 6.

Key points

- Doctest is strongest when a small deterministic transcript is already useful documentation.
- Check attempted and failed counts so an empty collection cannot masquerade as success.
- Read input, expected, got, and intended contract before editing a failure.
- Use whitespace, ellipsis, and exception options only for narrow variation that is not part of the public promise.
- Prefer stable public views over timestamps, addresses, random values, or unordered representations.
- Move complex setup and boundary tables to pytest when structured assertions communicate better.

References

- Python `doctest` documentation
- Pytest doctest integration
- Python `repr()` documentation

Next: Review Documentation Like Code

Review Documentation Like Code

Online lesson and runnable notebook

Build a documentation review gate that executes public examples, resolves local links, inspects rendered pages, and still leaves reader judgment visible.

- **Level:** Beginner
- **Estimated time:** 2.5–3.5 hours
- **You will learn:** Turn reader promises into focused checks, diagnose false-green documentation gates, and combine automation with human review.
- **Practice in:** A local project containing README, QMD, Python, tests, and rendered HTML

Documentation changes with code. A renamed function can break a quick start; a moved page can break navigation; a revised output can leave a doctest stale; a successful render can still publish an unusable paragraph. Treat documentation as reviewable source with evidence, while remembering that automation cannot become the reader.

This lesson asks:

1. Which reader promises can become observable checks?
2. How can a check prove it inspected the intended files and examples?
3. What makes a broken-link report actionable?
4. Which rendered-site facts should be inspected after a successful build?
5. Which qualities still require a fresh reader and accessibility review?

1. Translate promises into checks of the right strength

Begin with a documentation promise, not a tool:

Reader promise	Useful automated evidence	Important remaining judgment
The quick start ranks Moon Dial	Public code executes and returns score 44	Is the setup understandable to a newcomer?
Two docstring examples are current	Two attempts, zero failures	Are those examples worth showing?
README links reach deeper pages	Every local target file exists	Are labels and destinations useful?
The site contains API reference	Clean render creates `api.html`	Can a caller find the needed fact quickly?
The page is accessible to navigate	Headings, link text, and alt attributes are inspected	Does the complete experience work with relevant assistive technology?

A check is strongest when it names the promise and fails with evidence that helps repair it. It is weakest when it counts an unrelated token and declares the document good.

This test protects the ordinary public result:

```python
from museum_quest import ExhibitScore, rank_exhibit


def test_readme_quick_start_result() -> None:
    assert rank_exhibit("Moon Dial", votes=4, minutes_open=75) == ExhibitScore(
        name="Moon Dial",
        score=44,
    )
```

It does not prove the README contains that code or tells the reader how to install the package. Add a source-level assertion only when the ownership is clear:

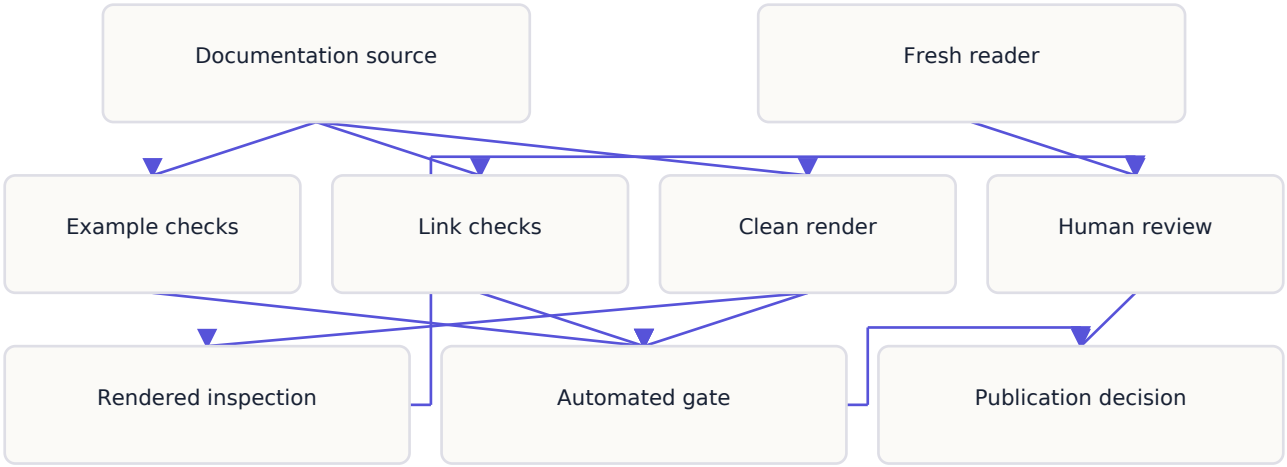
```python
from pathlib import Path

ROOT = Path(__file__).parents[1]


def test_readme_names_the_public_entry_path() -> None:
    readme = (ROOT / "README.md").read_text(encoding="utf-8")
    assert "## Requirements" in readme
    assert "## Quick start" in readme
    assert "from museum_quest import rank_exhibit" in readme
    assert "ExhibitScore(name='Moon Dial', score=44)" in readme
```

String checks are intentionally narrow. They can detect a removed section or stale result; they cannot judge whether the surrounding instructions are clear. Do not create hundreds of brittle assertions over exact prose.

Documentation trust comes from several evidence sources; no single green check covers the whole reader experience.



2. Make every check prove it inspected something

A false-green gate often has correct-looking assertions over an empty set:

```python
from pathlib import Path
```

PYTHON (CONTINUED)

```
pages = list(Path("documentation").glob("*.qmd"))
assert all("title:" in page.read_text() for page in pages)
```

If the real folder is `docs/`, `pages` is empty. Python's `all([])` is `True`. The gate inspected nothing and passed.

Protect collection first:

PYTHON

```
from pathlib import Path

pages = sorted(Path("docs").glob("*.qmd"))
assert pages, "no QMD pages found under docs/"
assert all("title:" in page.read_text(encoding="utf-8") for page in pages)
```

The same principle applies to doctest attempts, rendered HTML paths, link matches, and selected navigation entries. Record an expected count or named set when the project has a stable contract.

Run from a stable root

Checks should not depend on the terminal's accidental directory:

PYTHON

```
from pathlib import Path

PROJECT_ROOT = Path(__file__).resolve().parents[1]
DOCS_ROOT = PROJECT_ROOT / "docs"
README = PROJECT_ROOT / "README.md"
```

A test file can derive paths from its own location. A command-line script should accept or deliberately locate a root and print it in failure output.

Checkpoint: reject empty evidence

Make the gate prove collection

Choose the assertion that prevents an empty inspection from passing.

1. Why can `all(check(page) for page in pages)` pass when the path is wrong?

- **A.** `all` returns true for an empty iterable
- **B.** Path objects always pass checks
- **C.** QMD files cannot be read
- **D.** Assertions ignore generators

2. Which doctest evidence rules out silent non-collection?

- **A.** Zero failures only
- **B.** The intended attempted count plus zero failures
- **C.** A docstring length greater than zero
- **D.** One imported module

3. What makes a documentation test less dependent on the shell's current directory?

- **A.** Use the author’s absolute home path
- **B.** Search the whole disk
- **C.** Derive the project root from the test file’s resolved path
- **D.** Assume every command starts under `docs/`

Write your choices before checking the answer key at the end of this book.

3. Report broken links with source and target

A README link has two relevant paths: the source document and the destination resolved relative to that source. This small checker handles simple local Markdown links:

```python
import re
from pathlib import Path
from urllib.parse import unquote

MARKDOWN_LINK = re.compile(r"\[[^\]]+\]\(([^)]+)\)")


def local_targets(page: Path) -> list[Path]:
    """Return local file targets from simple inline Markdown links."""
    targets = []
    text = page.read_text(encoding="utf-8")
    for raw_target in MARKDOWN_LINK.findall(text):
        target_without_fragment = raw_target.split("#", 1)[0]
        if not target_without_fragment:
            continue
        if "://" in target_without_fragment or target_without_fragment.startswith(
            "mailto:"
        ):
            continue
        decoded = unquote(target_without_fragment)
        targets.append((page.parent / decoded).resolve())
    return targets


def missing_local_links(pages: list[Path]) -> list[tuple[Path, Path]]:
    """Return source/target pairs for missing local files."""
    missing = []
    for page in pages:
        for target in local_targets(page):
            if not target.exists():
                missing.append((page, target))
    return missing
```

Use it with an explicit source set:

```python
from pathlib import Path

root = Path(".").resolve()
pages = [root / "README.md", *sorted((root / "docs").glob("*.qmd"))]
assert pages[0].is_file(), "README.md is missing"
assert len(pages) >= 6, f"expected README plus five docs pages, got {pages}"

missing = missing_local_links(pages)
assert missing == [], "missing local documentation links: " + repr(missing)
```

The failure includes both page and resolved target. A learner can tell whether the source used the wrong relative depth or the destination was removed.

This is a deliberately limited parser. It does not validate reference-style links, raw HTML, generated anchors, remote HTTP status, or whether a fragment exists in the target. State the limit rather than calling it a complete link checker. Quarto's render and project tests can cover additional known routes.

Do not make remote availability a fragile unit test

A live external site can be slow, unavailable, rate-limited, or intentionally redirected. Network checks also make offline learning fail. Keep descriptive external links under review, but separate scheduled link monitoring from the fast deterministic local gate.

4. Execute examples from their documented context

Copying a README block into a test creates two sources. Instead, choose an ownership strategy:

1. **Executable example file:** README shows or links to `examples/quick_start.py`; tests run that file as a subprocess.
2. **Doctest source:** one docstring or text file owns the transcript, and the rendered documentation includes or faithfully reproduces it.
3. **Named public call:** a small test protects the exact public result while a human review ensures the README presents the same call clearly.

For a committed example script:

PYTHON
```
from museum_quest import rank_exhibit


def main() -> None:
    result = rank_exhibit("Moon Dial", votes=4, minutes_open=75)
    print(result)


if __name__ == "__main__":
    main()
```

A subprocess test checks the reader-facing output and clean process boundary:

PYTHON
```
import subprocess
import sys
from pathlib import Path

ROOT = Path(__file__).resolve().parents[1]


def test_quick_start_script() -> None:
    completed = subprocess.run(
        [sys.executable, "examples/quick_start.py"],
        cwd=ROOT,
        text=True,
        capture_output=True,
        check=False,
    )
    assert completed.returncode == 0, completed.stderr
    assert completed.stdout == (
        "ExhibitScore(name='Moon Dial', score=44)\n"
    )
```

Use the active interpreter through `sys.executable`. Set `cwd` explicitly. Keep `check=False` only because the test wants to assert and display both exit status and stderr itself.

A test run against an editable source checkout still does not prove published installation. If the README promises an installed wheel, add the Unit 10 clean wheel test at that boundary.

Checkpoint: choose honest source ownership

Keep reader examples and checks connected

Choose the test that observes the documentation promise rather than a substitute.

1. The README points readers to `examples/quick_start.py`. What is strong automated evidence?

- **A.** Count the word `example` in README
- **B.** Import a private helper instead
- **C.** Check only that the filename string appears
- **D.** Run the committed script in a clean process and compare its exit status and exact result

2. Why should the subprocess test set `cwd=ROOT`?

- **A.** To reproduce the documented project-root context
- **B.** To make all absolute paths portable
- **C.** To skip package imports
- **D.** To disable stderr

3. What risk appears when README, tutorial, and tests each copy a long example independently?

- **A.** Python refuses repeated text
- **B.** The copies can drift and checks may protect only one
- **C.** Quarto deletes all code blocks
- **D.** The public API becomes private

Write your choices before checking the answer key at the end of this book.

5. Inspect the rendered artifact, not only the exit code

A successful Quarto process means the renderer completed. It does not guarantee the site contains the intended pages or useful navigation.

After rendering, check named outputs:

```python
from pathlib import Path

site = Path("docs/_site")
expected_pages = {
    "index.html",
    "tutorial.html",
    "how-to-rank-exhibits.html",
    "why-ranking-is-deterministic.html",
```

PYTHON (CONTINUED)

```
    "api.html",
}

observed_pages = {path.name for path in site.glob("*.html")}
missing_pages = expected_pages - observed_pages
assert missing_pages == set(), f"missing rendered pages: {sorted(missing_pages)}"
```

Then inspect at least one rendered file with a deliberately narrow standard- library check:

PYTHON

```
from html.parser import HTMLParser


class HeadingCollector(HTMLParser):
    def __init__(self) -> None:
        super().__init__()
        self.in_heading = False
        self.headings: list[str] = []
        self._parts: list[str] = []

    def handle_starttag(
        self,
        tag: str,
        attrs: list[tuple[str, str | None]],
    ) -> None:
        if tag in {"h1", "h2", "h3"}:
            self.in_heading = True
            self._parts = []

    def handle_data(self, data: str) -> None:
        if self.in_heading:
            self._parts.append(data)

    def handle_endtag(self, tag: str) -> None:
        if self.in_heading and tag in {"h1", "h2", "h3"}:
            self.headings.append("".join(self._parts).strip())
            self.in_heading = False


collector = HeadingCollector()
collector.feed(Path("docs/_site/tutorial.html").read_text(encoding="utf-8"))
assert "Rank Your First Museum Exhibit" in collector.headings
```

This proves a named heading reached HTML. It does not replace opening the page, using keyboard navigation, checking narrow screens, or evaluating whether the heading sequence makes sense.

Treat warnings as evidence, not background noise

For every warning, ask:

- Which source file and line produced it?
- Does it affect a public page, link, execution result, or accessibility?
- Is it a known renderer limitation with a recorded reason?
- Would converting every warning to failure improve trust or only create noise?

This repository has a known non-fatal OJS block-count warning on pages with multiple quizzes. It should not license ignoring an unrelated broken-link or Mermaid syntax warning. Classify by source and effect.

6. Human review asks questions automation cannot settle

A fresh reader can reveal:

- a prerequisite that an experienced maintainer silently supplies;
- a term introduced after it is used;
- a safe but unnecessarily long route to the first result;
- a link whose target exists but answers another question;
- output that matches yet is not explained;
- a keyboard, contrast, screen-reader, or narrow-screen problem missed by a source check;
- a limitation that should appear before a risky action; or
- prose that is technically true and practically confusing.

Use a compact review pass:

Review lens	Question
Reader and task	Can I name who opens this page and what success means?
Preconditions	Can the reader check every prerequisite before acting?
Procedure	Does each step name action, context, and evidence?
Public truth	Do names, types, units, results, errors, and limits match code?
Navigation	Do headings and links reveal useful destinations?
Accessibility	Is meaning available without relying only on position, color, or an image?
Maintenance	Which code or route change should trigger review of this page?

Do not use “readability score passed” as a substitute for domain clarity. Automated prose metrics can flag a candidate sentence; a knowledgeable reader must decide whether technical meaning survives the revision.

7. Review the diff with the code change

Suppose a refactor renames private `_time_bonus` to `_score_complete_periods` while public behavior stays unchanged. Review:

- public README: probably unchanged;
- tutorial and how-to: unchanged if they use only `rank_exhibit`;
- API reference: unchanged if private helpers are omitted;
- public docstrings: unchanged unless they leaked the helper name;
- maintainer explanation: possibly changed if it discusses internal structure.

Now suppose the public parameter changes from `minutes_open` to `open_duration_minutes`. Review all public calls, signatures, docstrings, examples, API pages, how-to instructions, doctests, type references, and release notes. A check list derived from the public contract is more useful than "update the docs" as one final box.

Checkpoint: decide what green cannot prove

Combine automated and human evidence

Choose the review conclusion justified by the evidence.

1. A link checker confirms that `api.qmd` exists. What remains a human question?

- **A.** Whether the destination actually answers the reader's API question
- **B.** Whether the source file has bytes
- **C.** Whether `Path.exists()` returns a boolean
- **D.** Whether QMD uses text

2. Quarto exits zero and creates `tutorial.html`. What should still be inspected?

- **A.** Nothing; rendering proves all quality
- **B.** Only the file size
- **C.** Only the cache directory
- **D.** Navigation, headings, code presentation, links, and the actual reader path

3. A refactor renames only a private helper. Which documentation is most likely to need no change?

- **A.** A maintainer page that names the helper
- **B.** A private implementation diagram
- **C.** A README quick start that uses only the unchanged public function
- **D.** A source comment beside the helper

Write your choices before checking the answer key at the end of this book.

8. Lab: repair a false-green documentation gate

The museum project has five defects:

1. README promises score 46 for the 75-minute quick start.
2. The public docstring contains examples, but pytest collects no doctests.
3. `docs/tutorial.qmd` links to missing `reference/api.qmd`.
4. The tutorial title is `Details`.
5. A page-discovery check uses the nonexistent `documentation/` directory and passes over an empty list.

Create or repair `tests/test_documentation.py` so that it:

- derives the project root from `__file__`;

- asserts the named documentation source set is non-empty;
- protects the quick-start public result;
- checks doctest attempted and failed counts;
- reports each missing local link as `(source, target)`;
- renders the site through the documented command or verifies render output in a staged local sequence;
- asserts the five expected HTML filenames;
- leaves heading quality and reader-path review in an explicit human checklist.

Run the gate in this order:

BASH
```
pytest --doctest-modules src tests -q
quarto render docs
pytest tests/test_documentation.py -q
```

Record the earliest failure after each repair. Do not weaken score 44, remove the missing-link assertion, or accept zero doctest attempts merely to proceed.

Hint: protect the inventory before the details

Assert the README and five QMD source pages by name. Then a wrong directory fails as an inventory problem before `all()` can hide it.

Reveal a progressive gate outline

PYTHON
```
from pathlib import Path

ROOT = Path(__file__).resolve().parents[1]
DOCS = ROOT / "docs"
EXPECTED_SOURCE = {
    ROOT / "README.md",
    DOCS / "index.qmd",
    DOCS / "tutorial.qmd",
    DOCS / "how-to-rank-exhibits.qmd",
    DOCS / "why-ranking-is-deterministic.qmd",
    DOCS / "api.qmd",
}
EXPECTED_HTML = {
    "index.html",
    "tutorial.html",
    "how-to-rank-exhibits.html",
    "why-ranking-is-deterministic.html",
    "api.html",
}

assert all(path.is_file() for path in EXPECTED_SOURCE)
assert {path.name for path in (DOCS / "_site").glob("*.html")} >= EXPECTED_HTML
```

Combine this inventory with the public-result, doctest-count, and local-link checks developed earlier. Keep the human review table beside the automated gate instead of pretending these sets judge prose quality.

Key points

- Begin with a reader promise and choose evidence of matching strength.

- Assert collection or inventory before applying `all()` or counting failures.
- Resolve local links relative to their source and report both source and target.
- Execute examples from their documented process and directory; avoid drifting copies.
- Inspect expected rendered pages and important structure after a successful build.
- Human reader and accessibility review remain necessary even when every automated check passes.
- Review documentation according to the public effect of a change, not merely because any source line changed.

References

- Python `pathlib` documentation
- Python `html.parser` documentation
- Pytest good integration practices
- W3C tips for writing accessible content

Next: Build a Documentation Site with Quarto

Build a Documentation Site with Quarto

Online lesson and runnable notebook

Organize a multi-page Quarto website, configure reader-focused navigation and cross-references, diagnose render failures, and inspect the clean HTML artifact.

- **Level:** Beginner
- **Estimated time:** 3.5–5 hours
- **You will learn:** Turn QMD source and project configuration into a complete, navigable documentation site and repair failures using file, line, link, and output evidence.
- **Practice in:** A local Midnight Museum Quarto project; Colab is a reading companion rather than the site builder

A documentation site is not a pile of HTML files written by hand. Source pages, project configuration, navigation, assets, and a renderer combine into an output artifact. The source belongs in version control; the generated `_site` directory can be rebuilt and published after its checks pass.

This lesson asks:

1. Which files are source, configuration, cache, and public output?
2. How should navigation reflect reader tasks rather than alphabetical filenames?
3. Why should internal links point to QMD source paths?
4. When is preview useful, and why must deployment use a complete render?
5. How do you locate a failure in YAML, a page, a cross-reference, or generated output?

1. Separate source from rendered output

Use this learner-sized project:

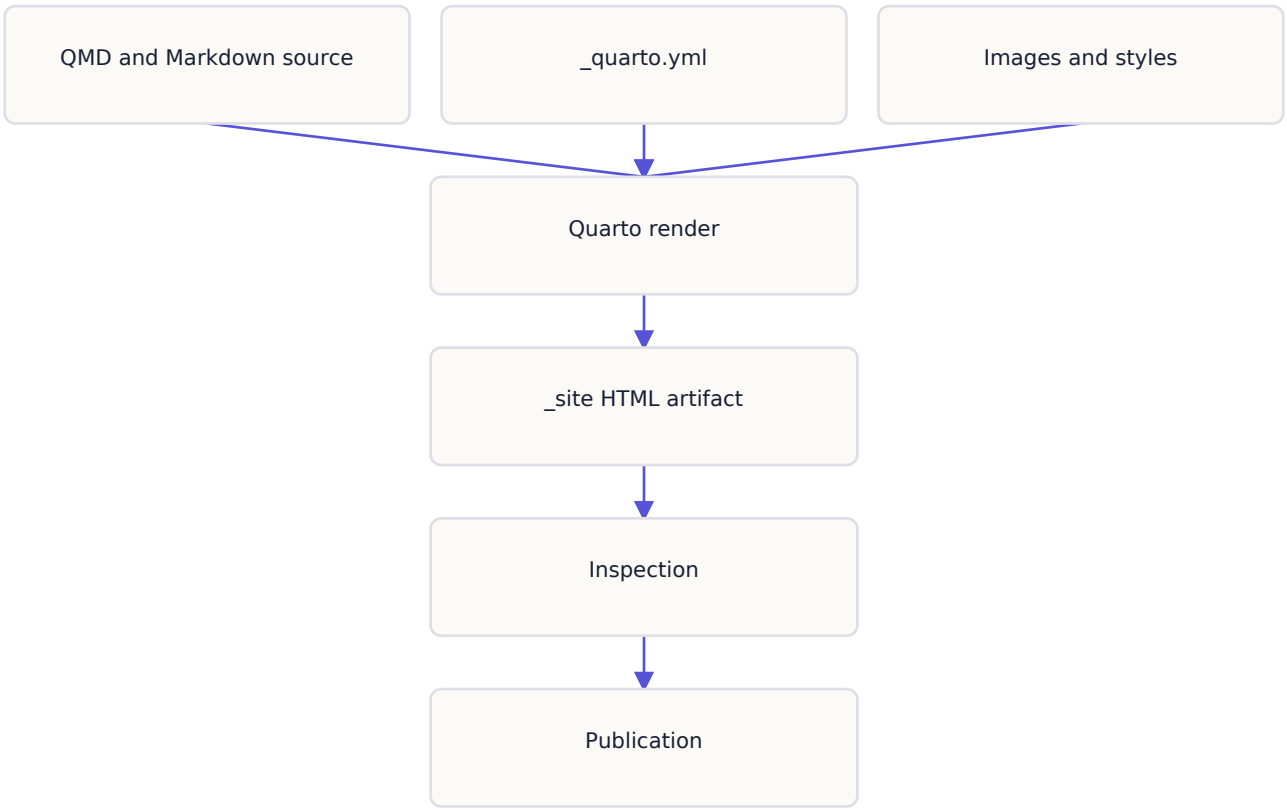
```text
midnight-museum/
├── README.md
├── pyproject.toml
├── src/museum_quest/
├── tests/
└── docs/
    ├── _quarto.yml
    ├── index.qmd
    ├── tutorial.qmd
    ├── how-to-rank-exhibits.qmd
    ├── why-ranking-is-deterministic.qmd
    ├── api.qmd
    └── images/
        └── reader-paths.svg
```

After a render, Quarto creates:

```text
docs/
├── .quarto/          # local project state/cache
└── _site/            # generated public artifact
    ├── index.html
    ├── tutorial.html
    ├── how-to-rank-exhibits.html
    ├── why-ranking-is-deterministic.html
    ├── api.html
    ├── search.json
    └── site_libs/
```

Do not hand-edit `_site/api.html` to fix source content. The next render will replace the edit. Find the owning QMD file or project configuration, repair it, and render again.

Quarto transforms reviewed source and configuration into a replaceable site artifact.



Create the smallest useful project file

Place `_quarto.yml` inside `docs/`:

```yaml
project:
  type: website
  output-dir: _site

website:
  title: Midnight Museum Quest
  page-navigation: true
  sidebar:
    contents:
```

YAML (CONTINUED)

```
      - index.qmd
      - tutorial.qmd
      - how-to-rank-exhibits.qmd
      - why-ranking-is-deterministic.qmd
      - api.qmd

format:
  html:
    toc: true
```

Read the fields by responsibility:

- `project.type` tells Quarto to build a website rather than unrelated documents.
- `output-dir` gives generated files one replaceable boundary.
- `website.title` labels the site.
- `page-navigation` adds previous/next movement through the declared order.
- `sidebar.contents` owns visible page sequence.
- `format.html.toc` gives longer pages local heading navigation.

YAML indentation carries structure. `contents` belongs under `sidebar`, which belongs under `website`. Use spaces consistently; do not use tabs.

Checkpoint: identify the owning file

Repair the source, not generated output

Choose the file or field that owns the observed behavior.

1. The API paragraph is wrong in `_site/api.html`. Where should the content repair normally be made?

- **A.** Inside the generated HTML only
- **B.** In `docs/api.qmd`, then rerender
- **C.** In the browser cache
- **D.** In `search.json`

2. The how-to appears before the tutorial in the sidebar. Which record owns this sequence?

- **A.** The Python function order
- **B.** The HTML file modification time
- **C.** `website.sidebar.contents` in `_quarto.yml`
- **D.** The README word count

3. Which directory should a clean render be able to replace completely?

- **A.** `src/`
- **B.** `tests/`
- **C.** `docs/images/`
- **D.** `docs/_site/`

Write your choices before checking the answer key at the end of this book.

2. Give each page an informative entrance

A minimal QMD page uses YAML front matter and meaningful headings:

````
MARKDOWN
---
title: Rank Your First Museum Exhibit
description: Follow a guided path from a clean project to one verified score.
---

## Before the museum opens

Use Python 3.10 or newer and complete the project installation.

## Rank the Moon Dial

```python
from museum_quest import rank_exhibit

result = rank_exhibit("Moon Dial", votes=4, minutes_open=75)
print(result)
```

Expected result: `ExhibitScore(name='Moon Dial', score=44)`.
````

The title becomes the page heading and default navigation label. The description can support previews and metadata. The page itself retains the reader-focused prerequisites, action, and expected result from Lesson 2.

Quarto features should clarify the task. A normal note uses a callout:

```
MARKDOWN

**Why 75 minutes adds four points**

Only complete 30-minute periods count. Seventy-five minutes contains two
complete periods.
:::
```

Do not put every paragraph into a colored box. Reserve callouts for information whose role—note, tip, warning, important point—helps a reader decide how to use it.

Link to source pages, not generated filenames

From `tutorial.qmd`, link to a sibling source:

```
MARKDOWN
[Look up the `rank_exhibit` contract](api.qmd "API reference")
```

Quarto resolves that source link to the appropriate output. A direct `api.html` link binds the source to one output format and is easier to break if the project later becomes a book or another format.

From root `README.md`, the same target uses its actual relative depth:

```
MARKDOWN
[Look up the `rank_exhibit` contract](docs/api.qmd "API reference")
```

Paths are relative to the page containing them, not to whichever terminal the author happens to use.

3. Design navigation around reader movement

The flat list works for five pages. Labels make the purpose even clearer:

```yaml
website:
  title: Midnight Museum Quest
  page-navigation: true
  search: true
  sidebar:
    style: floating
    contents:
      - href: index.qmd
        text: Start Here
      - href: tutorial.qmd
        text: Guided Museum Visit
      - href: how-to-rank-exhibits.qmd
        text: Change a Ranking
      - href: why-ranking-is-deterministic.qmd
        text: Why Rankings Repeat
      - href: api.qmd
        text: Public API
```

The order moves from entry to guided use, focused task, understanding, and exact lookup. Alphabetical order would begin with API, not the most useful entrance.

For a larger site, sections can group related pages:

```yaml
website:
  sidebar:
    contents:
      - href: index.qmd
        text: Start Here
      - section: Learn
        contents:
          - tutorial.qmd
      - section: Use the Package
        contents:
          - how-to-rank-exhibits.qmd
          - why-ranking-is-deterministic.qmd
          - api.qmd
```

Avoid creating a one-page section under every documentation type. A hierarchy should reduce choice, not make every click reveal another nearly empty menu.

Search helps when the site has enough content and readers know the term to find. It does not replace a clear entry journey for someone who does not yet know the function name.

Repository actions create feedback paths

A published project can expose edit and issue actions:

```yaml
website:
  repo-url: https://github.com/example/midnight-museum
  repo-actions:
    - edit
    - issue
```

Use the real repository URL. A feedback link without ownership or response expectations is not a complete support plan, but it gives readers an observable route to report stale content.

Checkpoint: organize the reader path

Make site navigation answer reader questions

Choose the configuration or link that preserves the intended path.

1. Which page should normally appear before API reference for a new museum learner?

- **A.** The guided museum tutorial
- **B.** An alphabetical list of private helpers
- **C.** Generated cache files
- **D.** The deployment workflow source

2. From `docs/tutorial.qmd`, which internal link best targets the sibling API source?

- **A.** The author's absolute `/home/.../api.qmd` path
- **B.** A descriptive source-relative link targeting `api.qmd`
- **C.** A `click` link targeting generated `api.html`
- **D.** A link that repeats the `docs/docs/` directory

3. What limitation does site search have?

- **A.** It cannot index text
- **B.** It always changes public APIs
- **C.** It cannot replace an entry path for readers who do not know the terms to search
- **D.** It prevents sidebars

Write your choices before checking the answer key at the end of this book.

4. Use cross-references when they reduce hunting

A figure with an identifier can be referenced by name:

```
MARKDOWN
![The README branches to tutorial, how-to, explanation, and API pages.](images/reader-paths.svg){#fig-r ↪
    eader-paths}

The four destinations in @fig-reader-paths answer different reader questions.
```

Quarto requires a type prefix such as `fig-`, `tbl-`, or `lst-`. Use hyphens in identifiers and keep them stable when other pages link to them.

A table can carry a caption and ID:

```
MARKDOWN
| Reader | Destination |
|---|---|
| New learner | Tutorial |
| Active caller | API reference |

: Reader questions choose documentation paths {#tbl-reader-paths}
```

Reference it as `@tbl-reader-paths`. The rendered number and link update if the table moves.

Do not cross-reference every short code block. A nearby sentence may be easier than “see Listing 14.” Use a reference when a reader needs to find or revisit a named object.

Diagnose an unresolved reference

Suppose the render warns that `@fig-reader_path` cannot be resolved. Compare the reference with the label:

```text
Reference: @fig-reader_path
Label:     #fig-reader-paths
```

There are two differences: underscore versus hyphen, and singular versus plural. Repair the source reference to the exact stable ID, then perform a full render and follow the link in HTML.

5. Preview quickly, render completely

From the `docs/` project directory:

```bash
quarto preview
```

Preview starts a local server and rerenders affected pages while you work. It is useful for checking headings, navigation, and layout in a browser. Stop the server before treating its current screen as a deployable artifact.

From the repository root, target the project explicitly:

```bash
quarto render docs
```

A successful full render reports an output such as:

```text
Output created: _site/index.html
```

When the project root is `docs/`, `_site` is normally `docs/_site`. Read paths in context instead of copying one output line into every project.

Why render again after preview?

- a page may never have been opened during preview;
- changes to `_quarto.yml`, shared includes, styles, or navigation affect more than the active page;
- caches can preserve state that a clean build will not have; and
- publication needs one complete artifact tied to reviewed source.

Clean generated output safely, rerender, and compare the named page inventory. Generated `_site` can be removed; QMD source cannot.

Checkpoint: choose preview or render

Use the build mode that fits the decision

Choose the command or evidence appropriate to the situation.

1. You changed shared navigation in `_quarto.yml` and are preparing deployment. What should you run?

- **A.** Open one cached HTML file
- **B.** Trust the last preview screen
- **C.** Render only an unrelated page
- **D.** A complete `quarto render` for the site

2. When is `quarto preview` especially useful?

- **A.** While iterating locally and inspecting page changes in a browser
- **B.** As proof that CI will install dependencies
- **C.** As a permanent hosting service
- **D.** To avoid the final complete render

3. Quarto exits zero. Which additional check protects the public page set?

- **A.** Count source comments
- **B.** Confirm each expected HTML path exists under `_site`
- **C.** Inspect only cache size
- **D.** Change the output directory after rendering

Write your choices before checking the answer key at the end of this book.

6. Read failures from the owning layer

Invalid YAML

```text
ERROR: YAMLException: bad indentation of a mapping entry
  at docs/_quarto.yml:9:7
```

Open the named file and line. Compare indentation with parent keys. Do not edit a QMD paragraph or generated HTML for a configuration parse error.

A common defect is:

```yaml
website:
  sidebar:
  contents:
    - index.qmd
```

`contents` must be nested beneath `sidebar`:

```yaml
website:
  sidebar:
    contents:
      - index.qmd
```

Missing source target

```
TEXT
Unable to resolve link target: reference/api.qmd
```

Start from the source page containing the link. Resolve `reference/api.qmd` relative to that page. Check whether the link depth is wrong, the file moved, or navigation still names a retired route.

Page parses but a diagram fails

A Mermaid error belongs to the diagram source. Keep labels simple, quote punctuation-sensitive labels, and preserve the required execution options. Run the focused Mermaid parser and render again rather than replacing the diagram with an unverified screenshot.

Render succeeds but a page is missing

Inspect project render targets and sidebar paths. A page excluded by project configuration will not reappear because its old HTML remains in an uncleared `_site`. This is why a clean output directory matters: stale files can conceal a missing render target.

7. Compare toolchains without maintaining three sites

Quarto is the complete path here because this course already uses QMD, code examples, diagrams, and a Quarto website. Other projects may choose differently:

- **Sphinx** has a mature Python domain and autodoc ecosystem for API-heavy libraries, often using reStructuredText or MyST.
- **MkDocs** emphasizes Markdown-oriented documentation websites and a plugin ecosystem.
- **Quarto** supports technical documents, executable content, cross-references, websites, books, and multiple output formats.

Choose from reader and maintainer needs: API generation, markup familiarity, execution model, extension ecosystem, output formats, accessibility, and team capacity. Do not build three versions before the project has one trustworthy site.

8. Lab: assemble the Midnight Museum site

Create the five-page source tree and `_quarto.yml` from this lesson. Your site must provide:

- `index.qmd` with value, prerequisites, and direct tutorial link;
- `tutorial.qmd` with the verified Moon Dial result;
- `how-to-rank-exhibits.qmd` for changing one input;
- `why-ranking-is-deterministic.qmd` explaining explicit inputs and no hidden state;
- `api.qmd` describing the public signature, result, and failures;
- reader-focused sidebar text and page navigation;
- one figure or table with a stable cross-reference;
- descriptive internal QMD links;
- repository feedback actions only if a real URL is supplied.

Run:

```bash
quarto render docs
```

Verify:

```python
from pathlib import Path

site = Path("docs/_site")
expected = {
    "index.html",
    "tutorial.html",
    "how-to-rank-exhibits.html",
    "why-ranking-is-deterministic.html",
    "api.html",
}
assert expected <= {path.name for path in site.glob("*.html")}
```

Then introduce and repair:

1. one incorrectly indented `contents` key;
2. one link to `reference/api.qmd` that should target `api.qmd`;
3. one cross-reference whose ID differs by an underscore;
4. one page removed from navigation while stale HTML remains.

Clear generated output before verifying the final repair so a stale file cannot produce a false pass.

Hint: build one navigable page before adding features

Start with `index.qmd`, `tutorial.qmd`, and a two-item sidebar. Render. Add one page and link at a time, rerendering after each structural change.

Reveal a complete minimal configuration

```yaml
project:
  type: website
  output-dir: _site

website:
  title: Midnight Museum Quest
  page-navigation: true
  search: true
  sidebar:
    style: floating
    contents:
      - href: index.qmd
        text: Start Here
      - href: tutorial.qmd
        text: Guided Museum Visit
      - href: how-to-rank-exhibits.qmd
        text: Change a Ranking
      - href: why-ranking-is-deterministic.qmd
        text: Why Rankings Repeat
      - href: api.qmd
        text: Public API

format:
  html:
    toc: true
```

The content of each page still owns reader quality. This configuration only makes the intended source set and order explicit.

Key points

- Keep QMD, configuration, and assets as source; treat `_site` as replaceable output.
- Design navigation around reader movement rather than filename order.
- Link to source QMD paths so Quarto can resolve the chosen output format.
- Use callouts and cross-references only when their structure helps the reader.
- Preview accelerates local iteration; a clean complete render owns the deployment artifact.
- Diagnose YAML, source-link, diagram, and stale-output failures at their owning layer.
- Teach and maintain one suitable toolchain completely before comparing alternatives.

References

- Creating a Quarto website
- Quarto website navigation
- Quarto cross-references
- Quarto callout blocks
- Sphinx documentation
- MkDocs documentation

Next: Publish Documentation and Keep It Trustworthy

Publish Documentation and Keep It Trustworthy

Online lesson and runnable notebook

Connect a clean local documentation contract to CI and GitHub Pages, diagnose deployment evidence, and maintain public promises as code and routes change.

- **Level:** Beginner
- **Estimated time:** 2.5–4 hours
- **You will learn:** Trace reviewed documentation through a clean CI build to one verified deployment artifact, then plan updates when public behavior or routes change.
- **Practice in:** A local Git project and a learner-sized GitHub Actions workflow

Publication changes the audience and the cost of mistakes. A local page can rely on an author's environment; a public site must be built from committed source in a clean runner, produce a named artifact, and expose the reviewed version. Even then, publication is a moment in an ongoing maintenance process—not proof that the documentation will remain current forever.

This lesson asks:

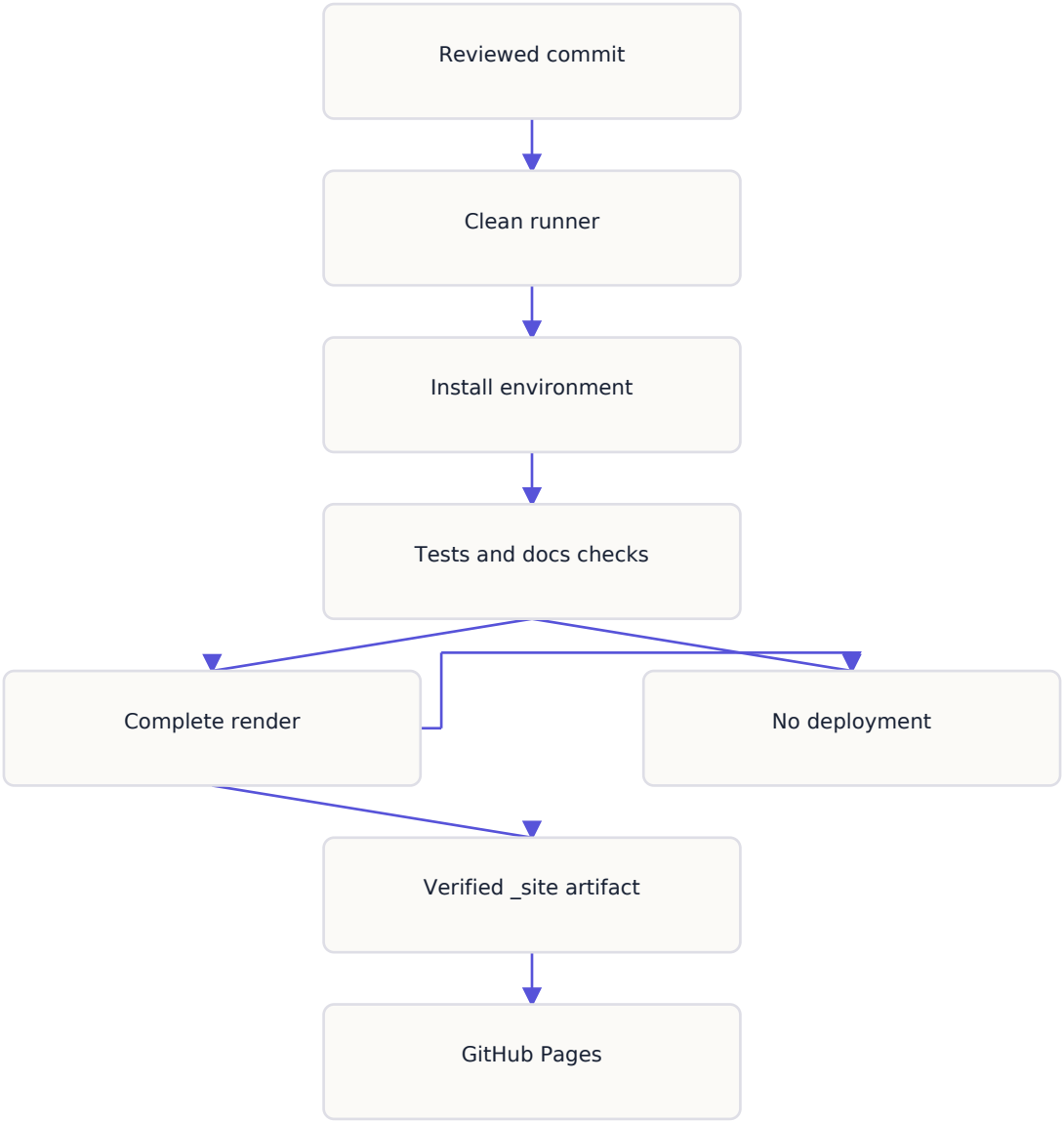
1. Which commit, environment, commands, and output directory produced the public site?
2. How does CI repeat the local documentation contract instead of inventing a second one?
3. What prevents a failed build from deploying stale or unrelated files?
4. How should a learner read setup, render, and deployment failures differently?
5. Which documentation changes follow a public API, support, or route change?

1. Trace one reviewed commit to one artifact

A useful publication path has explicit stages:

1. Check out the reviewed source commit.
2. Create the documented environment.
3. Install the project and documentation tools.
4. Run code and documentation checks.
5. Perform a complete Quarto render.
6. Verify the expected `_site` artifact.
7. Deploy that same artifact only when earlier stages succeeded.

The public site should be traceable to the reviewed commit and the artifact its successful gate produced.



Tests and docs checks → No deployment: failure

Complete render → No deployment: failure

The local contract should already work:

```bash
pytest --doctest-modules src tests -q
quarto render docs
python tests/check_rendered_docs.py
```

CI should repeat these responsibilities. If local instructions use Poetry while CI installs an unrelated requirements file, the two environments can drift. Share commands or keep their equivalence visible and tested.

Artifact means the generated site, not source

For the Midnight Museum configuration, Quarto writes under `docs/_site`. The workflow should publish that directory—not `docs/`, not repository root, not a stale `_site` created by an earlier job.

Verify it before deployment:

PYTHON

```python
from pathlib import Path

site = Path("docs/_site")
expected = {
    "index.html",
    "tutorial.html",
    "how-to-rank-exhibits.html",
    "why-ranking-is-deterministic.html",
    "api.html",
}

assert site.is_dir(), "Quarto output directory is missing"
observed = {path.name for path in site.glob("*.html")}
assert expected <= observed, f"missing published pages: {sorted(expected - observed)}"
```

This check cannot judge the prose, but it prevents a workflow from publishing an empty or incomplete page set under a successful deployment step.

2. Read a learner-sized documentation workflow

The repository’s real workflow includes environment setup, Poetry, the complete site build, and a GitHub Pages action. A smaller project can begin here:

YAML

```yaml
name: Documentation

on:
  pull_request:
  push:
    branches: [main]
  workflow_dispatch:

permissions:
  contents: write

concurrency:
  group: docs-${{ github.ref }}
  cancel-in-progress: true

jobs:
  build:
    runs-on: ubuntu-latest
    timeout-minutes: 15

    steps:
      - name: Check out reviewed source
        uses: actions/checkout@v4

      - name: Set up Python
        uses: actions/setup-python@v5
        with:
          python-version: "3.13"

      - name: Install project and test tools
        run: |
          python -m pip install --upgrade pip
          python -m pip install -e ".[dev]"

      - name: Test code and documentation examples
        run: pytest --doctest-modules src tests -q

      - name: Render the documentation site
        run: quarto render docs
```

YAML (CONTINUED)

```yaml
      - name: Verify the rendered page set
        run: python tests/check_rendered_docs.py

      - name: Publish the verified site
        if: ${{ github.event_name == 'push' && github.ref == 'refs/heads/main' }}
        uses: peaceiris/actions-gh-pages@v4
        with:
          github_token: ${{ secrets.GITHUB_TOKEN }}
          publish_branch: gh-pages
          publish_dir: docs/_site
          force_orphan: true
```

This example assumes Quarto is available on the runner and the project defines a `dev` extra. A complete workflow must install Quarto explicitly or use an environment/action that supplies it. Never copy setup lines without checking those assumptions.

Read each boundary:

- Pull requests build and review but do not publish.
- Pushes to `main` build and can publish after success.
- `workflow_dispatch` allows a deliberate manual build; its publish condition as written excludes it.
- A job stops after a failing step because later commands are not reached.
- The deployment step's `if` narrows publishing even when builds run elsewhere.
- `publish_dir` points to the output verified immediately before it.
- Concurrency cancels an older in-progress publication for the same ref.

Permissions deserve the smallest scope supported by the selected publishing method. This action pushes a branch, so the example grants `contents: write`. Other official GitHub Pages workflows use artifact upload and Pages-specific permissions instead. Follow the current documentation for the method you adopt.

Checkpoint: trace build and deploy

Connect the reviewed source to the public artifact

Choose the workflow decision that preserves traceability.

1. Quarto renders the museum site to `docs/_site`. Which directory should the deployment step publish?

- **A.** Repository root
- **B.** `src/`
- **C.** `docs/_site`
- **D.** The runner cache

2. Why should the pull-request run normally build without publishing?

- **A.** Pull requests cannot run tests
- **B.** HTML cannot be generated in CI
- **C.** Review needs only source formatting

- **D.** It verifies proposed source without replacing the public site before merge

3. What reduces drift between local and CI documentation checks?

- **A.** Run the same responsibilities and keep command/environment differences explicit
- **B.** Use unrelated tools in each environment
- **C.** Skip doctest in CI
- **D.** Publish a developer's existing `_site`

Write your choices before checking the answer key at the end of this book.

3. Preserve exit status through every command

A shell step can hide a failure:

```bash
quarto render docs || echo "render failed"
echo "continuing"
```

The `echo` after `||` succeeds, so the step may exit zero and allow deployment. Logging is not failure handling. Let the command fail, or capture status and exit nonzero after recording useful evidence.

Pipes can also conceal an earlier failure in shells that report only the final command:

```bash
quarto render docs | tee render.log
```

GitHub Actions' Bash environment normally uses failure-sensitive settings, but do not rely on memory when translating the command to another runner. A simple un-piped render is clearest for the learner workflow. If a log pipeline is necessary, configure and test pipe failure behavior.

Do not combine every responsibility into one opaque line:

```bash
pytest -q && quarto render docs && python tests/check_rendered_docs.py
```

The chain stops correctly, but separate named workflow steps make it clearer whether code examples, rendering, or artifact inspection failed.

A stale artifact can survive an incomplete build

If a runner reuses a workspace and the new render omits `api.qmd`, an old `api.html` may remain. Clean runners reduce this risk; explicit output cleanup and a named source/output inventory make it visible locally.

A deployment job should download the artifact created by the build job for the same workflow run—not rebuild a different commit or publish a path left on a persistent machine.

4. Read the earliest useful CI evidence

Environment failure

```text
ModuleNotFoundError: No module named 'museum_quest'
```

If it appears during doctest collection, inspect installation and working directory before changing a docstring result. Confirm the checked-out project was installed and the expected Python environment is active.

Missing renderer

```text
quarto: command not found
```

The workflow never reached QMD content. Install Quarto through the selected setup method and pin/record an appropriate version. Do not edit navigation to repair a missing executable.

Source/configuration failure

```text
ERROR: YAMLException: bad indentation of a mapping entry
  at docs/_quarto.yml:14:7
```

Now the renderer exists and has named a project file/line. Repair YAML structure, validate it locally, and run the complete gate.

Artifact failure

```text
AssertionError: missing published pages: ['api.html']
```

The render command returned zero, but the named output inventory is incomplete. Inspect source inclusion, navigation/render targets, and stale-output cleanup. Do not delete `api.html` from the expected set unless the public map was intentionally changed and reviewed.

Deployment failure

```text
remote: Permission to repository denied
```

The artifact may be correct while the publishing token or permissions are not. Inspect the deployment method’s required permission and branch settings. Do not add a personal token to committed YAML or lesson output.

Checkpoint: repair the owning layer

Locate the earliest failing boundary

Choose the repair that matches the observed layer.

1. CI reports `quarto: command not found`. What should be repaired first?

- **A.** The API expected score
- **B.** The runner’s Quarto installation/setup
- **C.** A README heading
- **D.** The published domain name

2. The render exits zero, but the artifact check reports missing `api.html`. What evidence should you inspect?

- **A.** Only token permissions
- **B.** Only Python version
- **C.** Whether `api.qmd` is included in the clean render and output inventory
- **D.** The author's browser history

3. Tests and render pass, but branch publication is denied. What should not be committed as a shortcut?

- **A.** The QMD source
- **B.** The verified output path
- **C.** The workflow trigger
- **D.** A personal access token

Write your choices before checking the answer key at the end of this book.

5. Publication does not freeze the product

Documentation freshness follows public change. Use a change-impact table:

Change	Documentation to review
Private helper renamed	Maintainer explanation only if it names the helper
Public parameter renamed	README examples, tutorial, how-to, docstring, API reference, doctest, release guidance
Supported Python floor raised	README requirements, package metadata, setup workflows, troubleshooting
Expected score policy changed	Examples, tests, explanation, API contract, migration note
Public QMD route moved	Navigation, internal/external links, redirects or migration plan
Deployment directory changed	Quarto config, artifact check, workflow `publish_dir`

This repository is intentionally pre-release and does not preserve obsolete routes automatically. A published library with existing readers may require redirects, stable IDs, versioned pages, or a deprecation period. Decide from the actual compatibility requirement rather than treating either "keep everything" or "delete everything" as universal.

Separate current truth from history

The README and current API reference should describe supported behavior now. Release notes or a changelog describe what changed. Mixing years of historical instructions into the quick start makes current use harder and can lead readers to obsolete commands.

When several versions remain supported, label the documented version visibly and ensure examples are built against it. Multi-version hosting is beyond this Foundations unit, but hiding version scope is not.

Give readers a correction path

Useful feedback routes include:

- an “Edit this page” link to the source;
- an issue link with a documentation template;
- a support address or forum with stated scope;
- a named maintainer/owner for review; and
- a page timestamp only when the project can interpret and maintain it.

A “last updated” badge generated on every build can appear fresh while the content remains stale. Prefer version/commit traceability and reviewed public claims.

6. Keep a small publication record

For a release or merge, retain:

```text
Source commit:        4c72...
Python:               3.13
Quarto:               1.9.38
Doctest:              2 attempted, 0 failed
Pytest:               8 passed
Rendered pages:       5 expected, 5 present
Known warnings:       none
Published artifact:   docs/_site from this workflow run
Public smoke check:   index and API route loaded
```

The values are evidence, not decoration. Do not copy them into future records without rerunning. A smoke check of the hosted route catches hosting/path configuration that local HTML inspection cannot.

7. Compare publication methods by trust boundary

Quarto documents several ways to publish to GitHub Pages:

- render locally and commit generated output;
- use `quarto publish` from a trusted local environment; or
- render and deploy through CI.

This course prefers CI for the connected project because it ties clean setup, checks, rendering, and deployment to a reviewed commit. That preference is not a claim that one hosting method fits every organization. The important questions remain: who can publish, which source was built, which tools/versions ran, what artifact passed, and how a failure prevents replacement of the public site.

Checkpoint: maintain public trust

Update documentation with public change

Choose the maintenance action that follows the affected contract.

1. A public parameter is renamed. Which review scope is appropriate?

- **A.** Public examples, docstrings, API reference, task guides, tests, and change guidance that use the name
- **B.** Only a private helper comment
- **C.** Only generated HTML
- **D.** No documentation because types still pass

2. A widely published API page moves to a new route. What should a mature project decide explicitly?

- **A.** Whether the new filename sorts alphabetically
- **B.** Whether readers need a redirect or documented route migration
- **C.** Whether to delete every incoming link
- **D.** Whether to hide the page from search

3. Why is an automatically refreshed `last updated` date weak freshness evidence?

- **A.** Dates cannot appear in HTML
- **B.** Quarto rejects dates
- **C.** A rebuild can change the date without reviewing the content’s public claims
- **D.** It prevents source control

Write your choices before checking the answer key at the end of this book.

8. Lab: repair a workflow that publishes the wrong evidence

Audit this defective design:

```yaml
steps:
  - uses: actions/checkout@v4
  - name: Render
    run: quarto render docs || echo "ignored"
  - name: Publish
    uses: peaceiris/actions-gh-pages@v4
    with:
      github_token: plain-text-token
      publish_dir: docs
```

Problems include:

- no Python/Quarto environment setup;
- no package installation, pytest, doctest, links, or page inventory;
- a swallowed render failure;
- a committed secret placeholder that invites unsafe replacement;
- source `docs/` published instead of generated `docs/_site`;
- no branch/event condition preventing pull-request publication;
- no connection between a verified artifact and deployment.

Write a corrected learner-sized workflow using the staged design from section 2. Then simulate its local contract:

```bash
pytest --doctest-modules src tests -q
quarto render docs
python tests/check_rendered_docs.py
```

Create one failure at a time:

1. remove the package installation and observe the import boundary;
2. misspell `sidebar.contents` and observe the render boundary;
3. remove `api.qmd` from the source set and observe the artifact boundary;
4. set `publish_dir: docs` and explain why a successful deployment would still publish the wrong thing.

Record each as environment, check, render, artifact, or deployment evidence.

Hint: make publication the final consumer

Draw arrows from checkout through setup, tests, render, and artifact inspection. The publish step should have only one incoming artifact path and no way to run after an earlier failure.

Reveal the critical corrected boundaries

```yaml
- name: Test code and documentation examples
  run: pytest --doctest-modules src tests -q

- name: Render documentation
  run: quarto render docs

- name: Verify rendered pages
  run: python tests/check_rendered_docs.py

- name: Publish verified site
  if: ${{ github.event_name == 'push' && github.ref == 'refs/heads/main' }}
  uses: peaceiris/actions-gh-pages@v4
  with:
    github_token: ${{ secrets.GITHUB_TOKEN }}
    publish_dir: docs/_site
```

This fragment still depends on earlier checkout and environment setup. Keep the complete workflow readable from top to bottom rather than publishing only the final fragment.

Key points

- Trace the public site to one reviewed commit, clean environment, complete gate, and named generated artifact.
- Reuse the local documentation contract in CI and keep unavoidable environment differences explicit.
- Never swallow the exit status of tests or rendering before deployment.
- Read setup, source, render, artifact, and permission failures at their owning boundaries.
- Publish the verified `_site` directory, not source, cache, or an artifact from another run.

- Review documentation according to public change and make route/version compatibility decisions explicitly.
- Publication creates a feedback and maintenance responsibility; it does not make content permanently current.

References

- Quarto publishing to GitHub Pages
- GitHub Actions workflow syntax
- GitHub Pages custom workflows
- Quarto website workflow

Next: Publish the Puzzle Garden Field Guide

Unit Challenge: Publish the Puzzle Garden Field Guide

Online lesson and runnable notebook

Repair and publish the documentation for a playful clue-path package so a new player can reveal, customize, understand, and verify a garden route.

- **Level:** Python Foundations · Unit 15 challenge
- **Estimated time:** 60–120 minutes
- **Outcome assessed:** Design, write, test, build, and publish documentation that guides real readers from first contact to successful use, explains a stable public Python API, and remains trustworthy as the code changes.
- **Evidence:** Reader-focused source, executable examples, resolved links, clean rendered pages, a publication gate, and one debugging record

1. Reopen the Puzzle Garden

The Puzzle Garden’s code still works, but its field guide does not. The quick start calls a retired function, the expected path is stale, the public docstring hides an invalid-clue failure, two pages are missing from navigation, and the workflow publishes source instead of the rendered site.

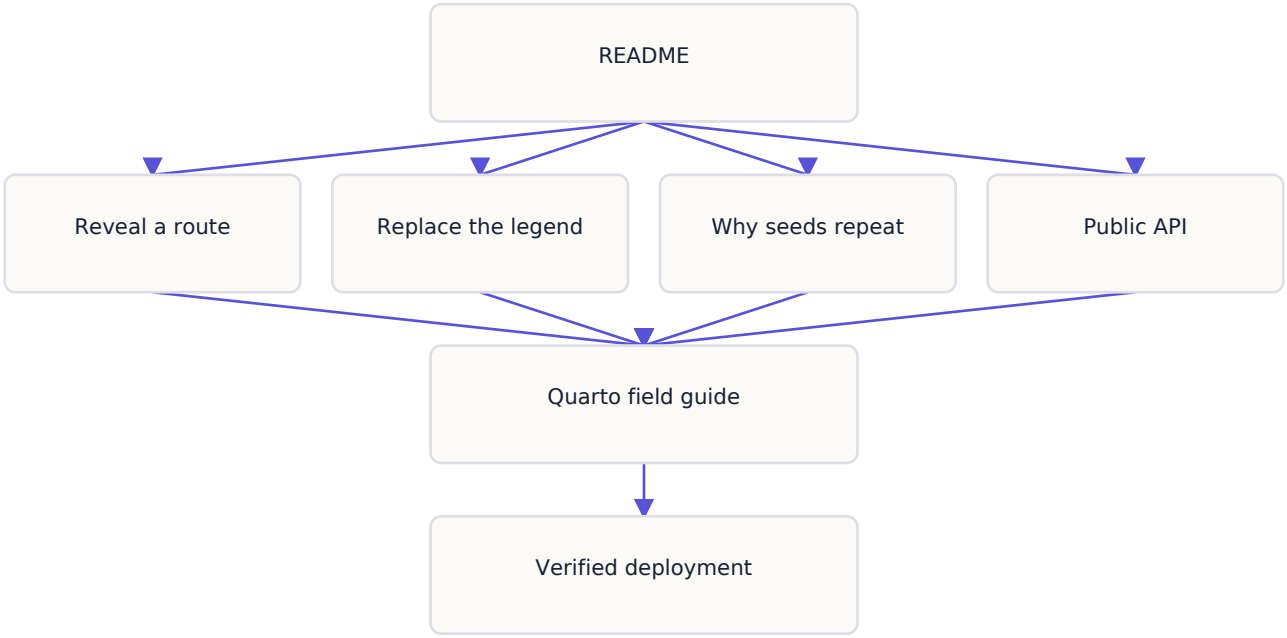
Your goal is to publish a small field guide that lets another player:

1. understand what the package reveals;
2. prepare the lesson project and reveal one exact path;
3. replace the clue legend for a new theme;
4. understand why the same seed repeats the same path;
5. look up the public call and anticipated failure; and
6. trust that the examples, links, pages, and deployment artifact were checked.

The satisfying result is visible. The quick start should reveal:

```text
('Moss arch', 'Blue gate', 'Lantern lilies')
```

Each reader path must reach a checked source page, and every page must reach the same rendered field guide artifact.



Use the scaffold to avoid blank-page work

The package behavior, file names, public signature, initial page headings, and progressive checks are supplied. Your central work is deciding what each reader needs, repairing public promises, and connecting the checked artifact to publication. :::

2. Inspect the supplied project

Begin with this tree:

```text
puzzle-garden/
├── README.md
├── pyproject.toml
├── src/puzzle_garden/
│   ├── __init__.py
│   └── trail.py
├── tests/
│   ├── test_trail.py
│   └── test_documentation.py
├── docs/
│   ├── _quarto.yml
│   ├── index.qmd
│   ├── first-route.qmd
│   ├── customize-legend.qmd
│   ├── why-seeds-repeat.qmd
│   └── api.qmd
└── .github/workflows/docs.yml
```

The public package implementation already passes its ordinary unit tests. Do not redesign its algorithm. Documentation changes may improve the public module/function docstrings, examples, README, QMD pages, checks, and workflow.

The source uses three clue symbols:

Symbol	Default location
`L`	Lantern lilies
`M`	Moss arch
`B`	Blue gate

A seed rotates the explicit clue sequence. It does not use global randomness or remember an earlier call.

Acceptance example

```python
from puzzle_garden import reveal_path

route = reveal_path("L M B", seed=1)
assert route == ("Moss arch", "Blue gate", "Lantern lilies")
print(route)
```

Changed legend:

```python
from puzzle_garden import reveal_path

moon_legend = {
    "L": "Lunar pond",
    "M": "Meteor arch",
    "B": "Blue moon gate",
}
assert reveal_path("L M B", legend=moon_legend) == (
    "Lunar pond",
    "Meteor arch",
    "Blue moon gate",
)
```

Important failure:

```python
from puzzle_garden import reveal_path

try:
    reveal_path("L X")
except ValueError as error:
    assert str(error) == "unknown clue: X"
else:
    raise AssertionError("unknown clue was accepted")
```

3. Start from the contract

Create `src/puzzle_garden/trail.py` from this supplied implementation. Complete its docstring; do not change the observable algorithm to make stale prose pass.

```python
from collections.abc import Mapping

DEFAULT_LEGEND = {
    "L": "Lantern lilies",
    "M": "Moss arch",
    "B": "Blue gate",
```

PYTHON (CONTINUED)

```
}


def reveal_path(
    clues: str,
    *,
    seed: int = 0,
    legend: Mapping[str, str] = DEFAULT_LEGEND,
) -> tuple[str, ...]:
    """Return decoded clue names rotated by a repeatable seed.

    TODO: Document parameter meaning, returned order, empty/unknown clue
    failures, and two deterministic examples.
    """
    symbols = clues.split()
    if not symbols:
        raise ValueError("clues cannot be empty")
    for symbol in symbols:
        if symbol not in legend:
            raise ValueError(f"unknown clue: {symbol}")
    offset = seed % len(symbols)
    ordered = symbols[offset:] + symbols[:offset]
    return tuple(legend[symbol] for symbol in ordered)
```

Export the public names in `src/puzzle_garden/__init__.py`:

PYTHON

```
from .trail import DEFAULT_LEGEND, reveal_path

__all__ = ["DEFAULT_LEGEND", "reveal_path"]
```

The starter README currently contains these defects:

MARKDOWN

```
# Garden Tool

Install it normally. Run `make_path("L M B")` and expect
`('Lantern lilies', 'Moss arch', 'Blue gate')`.

[Click here](docs/guide.qmd "retired guide") for more.
```

The starter Quarto sidebar lists only `index.qmd`, `first-route.qmd`, and `api.qmd`. The starter workflow ends with `publish_dir: docs`.

4. Build the field guide in small stages

Stage A — Repair the project entrance

Write a README with:

- a specific purpose and Python 3.10+ context;
- a truthful preparation/install path for the supplied source project;
- a complete public import and seed-1 quick start;
- the exact Moss arch → Blue gate → Lantern lilies result;
- limitations and support sections;
- descriptive relative links to the four deeper reader pages.

Stage B — Give every page one job

Finish these pages:

- `first-route.qmd`: a guided path using supplied clues and seed;
- `customize-legend.qmd`: a focused task for replacing every used symbol;
- `why-seeds-repeat.qmd`: explanation of rotation over explicit input and no hidden state;
- `api.qmd`: exact signature, parameters, return, and `ValueError` cases.

Use informative titles, visible prerequisites, expected output, meaningful link text, and recovery close to the failure it repairs.

Stage C — Make examples executable

Complete the public docstring with one ordinary example and the unknown-clue failure. Ensure doctest reports **2 attempted, 0 failed**. Keep the custom-legend boundary in pytest where a structured tuple assertion is clearer.

Stage D — Build and connect the site

Repair `_quarto.yml` so all five QMD pages appear in reader order. Render from a clean output directory and verify all five HTML files. Repair every local source link before publication.

Stage E — Publish only checked output

The workflow should run ordinary tests and doctests, render the complete site, verify the HTML inventory, and publish `docs/_site` only after success on the accepted branch. Never commit a personal token.

5. Run progressive assertions

Run these stages in order. Keep the expected values unchanged unless you can explain an approved contract change.

Check 1 — Public behavior

PYTHON

```
from puzzle_garden import reveal_path

assert reveal_path("L M B", seed=1) == (
    "Moss arch",
    "Blue gate",
    "Lantern lilies",
)
assert reveal_path("L M", seed=0) == (
    "Lantern lilies",
    "Moss arch",
)

try:
    reveal_path("L X")
except ValueError as error:
    assert str(error) == "unknown clue: X"
else:
    raise AssertionError("unknown clue was accepted")
```

Check 2 — Discovered examples

```python
import doctest
import puzzle_garden.trail

results = doctest.testmod(puzzle_garden.trail, verbose=False)
assert results.attempted == 2
assert results.failed == 0
```

Check 3 — Real README and source pages

```python
from pathlib import Path

root = Path(".").resolve()
readme = (root / "README.md").read_text(encoding="utf-8")

for heading in (
    "## Requirements",
    "## Installation",
    "## Quick start",
    "## Limitations",
    "## Support",
    "## Learn more",
):
    assert heading in readme, f"README missing {heading}"

assert "from puzzle_garden import reveal_path" in readme
assert "seed=1" in readme
assert "('Moss arch', 'Blue gate', 'Lantern lilies')" in readme

source_pages = {
    "index.qmd",
    "first-route.qmd",
    "customize-legend.qmd",
    "why-seeds-repeat.qmd",
    "api.qmd",
}
assert source_pages == {path.name for path in (root / "docs").glob("*.qmd")}
```

Check 4 — Local README targets

```python
from pathlib import Path

root = Path(".").resolve()
targets = {
    root / "docs/first-route.qmd",
    root / "docs/customize-legend.qmd",
    root / "docs/why-seeds-repeat.qmd",
    root / "docs/api.qmd",
}
assert all(path.is_file() for path in targets), sorted(targets)
```

Then run the broader source-link checker from Lesson 5 so links inside QMD pages are also resolved relative to their source.

Check 5 — Clean rendered artifact

```bash
quarto render docs
```

```python
from pathlib import Path

site = Path("docs/_site")
expected_html = {
    "index.html",
    "first-route.html",
    "customize-legend.html",
    "why-seeds-repeat.html",
    "api.html",
}
observed_html = {path.name for path in site.glob("*.html")}
assert expected_html <= observed_html, sorted(expected_html - observed_html)
```

Check 6 — Local/CI contract

```text
pytest --doctest-modules src tests -q
quarto render docs
python tests/check_rendered_docs.py
```

Confirm the workflow contains the same responsibilities in this order and that its deployment input is `docs/_site`.

A list of headings is not a README check

Do not replace file inspection with `README_HEADINGS = [...]` and then assert the list against itself. The checks must read the artifact the learner will publish. :::

6. Use the hint ladder only when needed

Hint 1

Anchor the README on the seed-1 route. State the project-root context, import `reveal_path`, show the exact tuple, and link each later reader question to its own page.

Hint 2

For the docstring, document `clues`, keyword-only `seed`, and `legend`; explain rotation and the returned tuple; include the ordinary call plus a traceback ending in `ValueError: unknown clue: X`. Verify attempted count as well as failures.

Hint 3

List all five QMD paths in `website.sidebar.contents`. Render from `docs/` or target the docs project from root. Publish `docs/_site` only after pytest, doctest, render, and page-inventory checks pass.

7. Keep debugging evidence

Preserve one real failing check before repairing it. Use the earliest useful evidence rather than the final symptom.

Failure	Exact evidence	One hypothesis	Controlled change	Verified rerun
What command/check failed?	Expected, observed, file/line/path	Which single cause fits?	What one source changed?	Which unchanged checks now pass?

Good candidates include:

- zero doctest attempts because the transcript prompt is malformed;
- expected default-order output while the quick start passes `seed=1`;
- `customize-legend.qmd` missing from the sidebar;
- a README link to retired `guide.qmd`;
- a clean render missing `api.html`; or
- workflow `publish_dir` pointing at source `docs` instead of `docs/_site`.

Do not use “changed everything and it worked” as the record. A reader should be able to connect the evidence to your one repair.

8. Compare with a complete solution path

Reveal after your checks pass or all three hints are exhausted

A complete public docstring can be:

PYTHON

```
from collections.abc import Mapping

DEFAULT_LEGEND = {
    "L": "Lantern lilies",
    "M": "Moss arch",
    "B": "Blue gate",
}


def reveal_path(
    clues: str,
    *,
    seed: int = 0,
    legend: Mapping[str, str] = DEFAULT_LEGEND,
) -> tuple[str, ...]:
    """Return decoded clue names rotated by a repeatable seed.

    Args:
        clues: Space-separated clue symbols.
        seed: Integer offset selecting the first clue in the returned route.
        legend: Mapping from clue symbols to visitor-facing location names.

    Returns:
        Decoded location names rotated by `seed`.

    Raises:
        ValueError: If `clues` is empty or contains an unknown symbol.

    Examples:
        >>> reveal_path("L M B", seed=1)
        ('Moss arch', 'Blue gate', 'Lantern lilies')
        >>> reveal_path("L X")
        Traceback (most recent call last):
        ...
        ValueError: unknown clue: X
    """
    symbols = clues.split()
    if not symbols:
        raise ValueError("clues cannot be empty")
    for symbol in symbols:
        if symbol not in legend:
            raise ValueError(f"unknown clue: {symbol}")
    offset = seed % len(symbols)
    ordered = symbols[offset:] + symbols[:offset]
```

PYTHON (CONTINUED)

```
    return tuple(legend[symbol] for symbol in ordered)
```

A complete README entry path can be:

MARKDOWN

````
# Puzzle Garden Field Guide

Reveal repeatable routes through a tiny symbolic garden puzzle.

## Requirements

Use Python 3.10 or newer and run commands from the project root.

## Installation

For this supplied source project, install in editable mode:

```text
python -m pip install -e .
```

## Quick start

```python
from puzzle_garden import reveal_path

print(reveal_path("L M B", seed=1))
```

Expected result: `('Moss arch', 'Blue gate', 'Lantern lilies')`.

## Limitations

Every clue must exist in the selected legend. The package stores no route state.

## Support

Report the clue string, seed, complete error, and Python version.

## Learn more

- [Reveal your first garden route](docs/first-route.qmd "first route")
- [Replace the clue legend](docs/customize-legend.qmd "customize legend")
- [Understand why seeds repeat](docs/why-seeds-repeat.qmd "seed explanation")
- [Look up the public API](docs/api.qmd "API reference")
````

The complete site configuration can be:

YAML

```
project:
  type: website
  output-dir: _site

website:
  title: Puzzle Garden Field Guide
  page-navigation: true
  search: true
  sidebar:
    contents:
      - href: index.qmd
        text: Start Here
      - href: first-route.qmd
        text: Reveal a Route
      - href: customize-legend.qmd
        text: Replace the Legend
```

YAML (CONTINUED)

```
      - href: why-seeds-repeat.qmd
        text: Why Seeds Repeat
      - href: api.qmd
        text: Public API

format:
  html:
    toc: true
```

Each QMD file still needs its own content. A concise complete set includes:

MARKDOWN

```
---
title: Reveal Your First Garden Route
---

## Prepare the project

Use Python 3.10+ and the editable installation from the README.

## Reveal the path

Run `reveal_path("L M B", seed=1)` and confirm the exact Moss arch, Blue gate,
Lantern lilies tuple.

## Recover from an unknown clue

Read the symbol named by `ValueError`, then add it to the legend or correct the
clue string.
```

MARKDOWN

```
---
title: Replace the Clue Legend
---

## Supply every used symbol

Create a mapping for `L`, `M`, and `B`; pass it through `legend=` and confirm the
returned tuple uses only the new location names.
```

MARKDOWN

```
---
title: Why the Same Seed Repeats a Path
---

The function rotates an explicit symbol list by `seed % len(symbols)`. Equal
clues, seed, and legend therefore produce the same tuple without global state.
```

MARKDOWN

```
---
title: Public API Reference
---

`reveal_path(clues, *, seed=0, legend=DEFAULT_LEGEND)` returns a tuple of decoded
names. It raises `ValueError` for empty clues or an unknown symbol.
```

Finally, the workflow’s gate and publication boundary should read clearly:

YAML

```
- name: Test code and documentation
  run: pytest --doctest-modules src tests -q
```

YAML (CONTINUED)

```yaml
- name: Render field guide
  run: quarto render docs

- name: Verify rendered pages
  run: python tests/check_rendered_docs.py

- name: Publish verified field guide
  if: ${{ github.event_name == 'push' && github.ref == 'refs/heads/main' }}
  uses: peaceiris/actions-gh-pages@v4
  with:
    github_token: ${{ secrets.GITHUB_TOKEN }}
    publish_dir: docs/_site
```

Compare reader outcomes, public contracts, checks, and artifact paths—not only sentence order. A different site structure is sound when it satisfies the same observable promises.

9. Check the field guide decisions

Puzzle Garden publication check

Choose the answer supported by your finished field guide and its evidence.

1. Which result proves that both intended public transcript examples were collected and passed?

- **A.** Zero failed without an attempted count
- **B.** The docstring contains text
- **C.** The package imports once
- **D.** Two attempted and zero failed

2. Where should a returning player learn only how to replace clue names?

- **A.** The focused legend how-to
- **B.** A repeated installation tutorial
- **C.** The deployment permission reference
- **D.** A private helper comment

3. Why does `reveal_path('L M B', seed=1)` begin at Moss arch?

- **A.** The function reads yesterday’s route
- **B.** The seed rotates the explicit symbols by one position
- **C.** Sets sort Moss first
- **D.** Quarto changes the tuple

4. What is the strongest local evidence for the README’s four deeper links?

- **A.** Their labels are all `click here`
- **B.** They worked in the author’s browser cache
- **C.** Each target resolves relative to README and the destination answers the named task
- **D.** The URLs are copied as plain text

5. Which directory should the workflow publish after a clean render and page check?

- **A.** `src/puzzle_garden`
- **B.** Repository root
- **C.** Source `docs`
- **D.** Generated `docs/_site`

Write your choices before checking the answer key at the end of this book.

10. Decide whether the challenge is complete

Use the evidence rather than page length:

Evidence	Ready to record when
Reader entrance	A clean learner can prepare the project and reproduce the seed-1 route using only README.
Page purposes	Tutorial, how-to, explanation, and API pages answer distinct questions and link meaningfully.
Public API	`help()`/docstring state inputs, rotation result, and both `ValueError` boundaries.
Executable examples	Exactly two intended doctest attempts pass; structured boundaries remain in pytest.
Site artifact	A clean render produces all five named HTML pages with repaired navigation and links.
Publication	CI repeats the local gate and publishes only the verified `_site` artifact.
Debugging	One record connects exact evidence to one hypothesis, controlled change, and clean rerun.

This button stores a self-reported marker only in this browser. It does not submit work, grade it, verify identity, or issue a certificate.

Key points

- Documentation is part of the Puzzle Garden result: it lets another player reveal and understand the route without the author’s hidden state.
- Progressive checks inspect real README, docstring, QMD, HTML, and workflow artifacts rather than stand-in lists.
- A reader-focused page, executable example, resolved link, clean render, and verified deployment each contribute different evidence.
- The strongest solution keeps the public API stable while repairing the promises around it.

Milestone Challenge: Publish a Tiny Community Library Tool

Online lesson and runnable notebook

Deliver a tested, installable command-line tool with documented behavior and reproducible checks.

- **Goal:** Deliver a tested, installable command-line tool with documented behavior and reproducible checks.
- **Estimated time:** 2–4 hours of active work.
- **Before starting:** Complete the milestone units and their unit challenges.
- **Keep:** Your source, observed outputs, checks, and a debugging note.

This community challenge is ungraded. Use the checks to review your own work; there is no submission service or automatic grade. Completion is self-reported in this browser on this device.

1. Publish a tiny community library tool

Model books by unique ID, title, and availability. Provide list, borrow BOOK_ID, and return BOOK_ID commands. Persist JSON at a configurable path. Unknown IDs, borrowing an unavailable book, returning an available book, and malformed storage must fail with a useful message and nonzero exit code. Failed operations must not overwrite storage. Separate model rules, argument parsing, and file access.

Predict an ordinary result before coding. Start with the smallest part you can run and inspect. Use Colab for notebooks and the local workspace introduced in the units for packaging or command-line tasks.

2. Set up the experiment

PYTHON

```
from dataclasses import dataclass

@dataclass
class Book:
    book_id: str
    title: str
    available: bool = True

# Add catalog rules, storage, and CLI modules.
```

The starter is deliberately incomplete. Implement the contract; do not change the expected results to match an incorrect implementation.

3. Check the observable results

1. Borrow, reload, and confirm unavailability; return, reload, and confirm availability.
2. For every rejected operation, assert an error status and unchanged stored bytes.
3. Use pytest temporary paths. Test model rules and at least one complete CLI invocation.

4 Run formatting, linting, type checking, and tests with documented commands.

5 Install in a clean environment and follow the README, including one executable documentation example.

Record actual outputs beside predictions. Reduce a failure to a small example, change one cause at a time, and repeat the failed check after repairing it.

4. Change one condition

Add an available-titles report without exposing storage internals. Begin with a failing test, make the smallest change, then explain the public interface before and after refactoring.

5. Review a hint after attempting the task

Hint: connect the relevant tools

Model state transitions first. Convert domain errors to CLI messages at the outer boundary. Validate before writing and inject storage paths in tests.

6. Explain and self-review

Deliver the package, tests, CI configuration, sample catalog, README, and a design note. Keep a failing test, repair, and passing result. Ask a peer to reproduce the instructions; this remains ungraded self-review.

Review path: compare your result with the contract

Check the ordinary result first, then the boundary cases and changed condition. Explain which rule justifies each result. An implementation that differs from the hint can still satisfy the contract; explain why yours does.

Check the reasoning behind your result

Choose an answer, then review the explanation.

1. Which test protects a rejected borrow operation?

- **A.** Check only that something prints.
- **B.** Test only the constructor.
- **C.** Use the shared real catalog.
- **D.** Check failure status and unchanged stored bytes.

Write your choices before checking the answer key at the end of this book.

7. Save your work and record completion

Save the notebook or project somewhere you control. Include one decision, one failed check, its repair, and a result you can reproduce. Record completion after reviewing the checks; you can undo it later.

Return to the milestone overview.

Checkpoint answer key

Try each checkpoint before reading its answers. Explanations are included so you can revisit the idea, rather than only check a letter.

Meet Python Through a Tiny Program

Source and output checkpoint

1. A. Source code is the written program that the interpreter attempts to run; output is evidence produced by that run.

2. B. The interpreter follows readable Python instructions and reports evidence; it does not infer missing requirements.

3. C. Python reaches the calls in source order, so the second call displays Ready as the second output line.

4. D. The quotes delimit the text in source code; print displays the contained text.

Controlled change checkpoint

1. A. The changed message belongs to one print call, so the controlled prediction affects that line only.

2. B. Python still displays the messages, but it reaches the reordered calls in their new top-to-bottom order.

3. C. That record connects one source change to a predicted and observed effect.

4. D. Exact behavior includes the content and position of every expected line.

First repair checkpoint

1. A. The source opens a quoted text value but does not close it, so pairing that delimiter addresses the evidence.

2. B. Valid source can still produce an unintended result; the exact output comparison reveals it.

3. C. Those details let another person reproduce the same boundary and evaluate the hypothesis.

4. D. The earliest failure may prevent later instructions from being reached, so repair and rerun it first.

Learn with Predictions and Evidence

Prediction and evidence checkpoint

1. A. The first prediction names observable content, order, and a boundary that the run can confirm or challenge.

2. B. The sentence reports directly visible evidence without yet claiming why it occurred.

3. C. The explanation points to the visible message in the source and does not invent unobserved behavior.

4. D. A pre-run record prevents hindsight from turning the observation into a pretend prediction.

Controlled experiment checkpoint

1. A. A controlled prediction names the intended effect and the boundaries that should stay stable.

2. B. Changing one relevant cause preserves a clear connection between the edit and the evidence.

3. C. The repair directly tests the smallest hypothesis supported by the earliest evidence.

4. D. Saved-file and path evidence can distinguish stale or wrong-file execution from a source-code defect.

Evidence and help checkpoint

1. A. The complete record shows how evidence justified a change and whether the change repaired the behavior.

2. B. Those details provide reproducible context without exposing unrelated or sensitive material.

3. C. Reduction and a preserved baseline create better evidence for the next investigation.

4. D. Editing source does not itself prove the resulting behavior; the focused rerun is the verification.

Use AI While Keeping the Thinking Yours

Bounded assistance checkpoint

1. A. The request asks for one diagnostic hint while leaving the repair and verification to the learner.

2. B. That packet gives reproducible context while limiting unrelated and sensitive material.

3. C. Your pre-run prediction preserves the reasoning practice and gives you a claim to compare.

4. D. Without a bounded problem, even a fluent response may solve the wrong task or obscure the useful next check.

AI claim verification checkpoint

1. A. The smallest local run directly challenges the named behavior and produces inspectable evidence.

2. B. Claim-by-claim review preserves useful information without importing the incorrect part.

3. C. The owning documentation and actual environment are better evidence than generated confidence.

4. D. Execution success and requirement success are separate claims.

Privacy and ownership checkpoint

1. A. A minimal invented case protects unrelated information and often makes the technical boundary clearer.

2. B. Formatting is not a safety review; understand the command’s effects and source before execution.

3. C. The record separates external suggestion from local evidence and the learner’s reasoning.

4. D. The learning outcome is critical verification, which the supplied cases support without a live service.

Work Reliably in Colab and Jupyter

Notebook parts checkpoint

1. A. The saved document and temporary execution process have different lifecycles.

2. B. Markdown cells make the reasoning and structure readable beside executable code.

3. C. Source edits and execution are separate actions, so visible output can become stale.

4. D. Hosted execution storage can be temporary; save important artifacts to a durable location.

Notebook state checkpoint

1. A. A fresh runtime knows only what has been executed in the current session.

2. B. Execution history controls current runtime state; visible position does not rewind the value.

3. C. Editing does not execute a cell, so the output may belong to the previous source.

4. D. A clean reader does not inherit the author’s unrecorded execution history.

Clean notebook checkpoint

1. A. An empty runtime exposes missing or out-of-order visible dependencies.

2. B. The first failing cell and its dependency provide the most direct evidence for repairing page order or source.

3. C. Output is part of the saved/shareable artifact and must be inspected independently.

4. D. Interrupting regains control without multiplying the same unknown work.

Build a Dependable Local Python Workspace

Python launcher checkpoint

1. A. One recorded working launcher gives later commands a consistent interpreter boundary.

2. B. Connecting pip to the chosen interpreter reduces mismatches between installation and execution.

3. C. The executable path lets you compare terminal, environment, editor, and notebook interpreter choices.

4. D. A restriction is an environment constraint to document and escalate, not a security control to bypass.

Files and editor checkpoint

1. A. The evidence points to a file path boundary rather than proving an interpreter installation failure.

2. B. A workspace folder makes file and interpreter context easier to inspect.

3. C. The terminal executes saved source, so unsaved editor state can explain the mismatch.

4. D. Different executable paths identify different Python environments with potentially different packages.

Virtual environment checkpoint

1. A. The environment isolates interpreter/package context; project source remains outside it.

2. B. Prompt decoration can be stale or customized, while interpreter paths identify the running process.

3. C. Activation is optional; the direct environment interpreter avoids changing security policy.

4. D. The comparison tests whether installation and execution use the same environment.

Unit Challenge: Launch Your Python Learning Lab

Launch round 1: first program

1. A. Each call displays its contained message when Python reaches it from top to bottom.

2. B. The prediction names both the intended effect and the stable boundaries of one controlled edit.

3. C. The opening quote has no partner; repairing only that delimiter tests the focused hypothesis.

4. D. Readable source can still produce the wrong result for the contract.

Launch round 2: assistance and evidence

1. A. One local call directly tests the named behavior without mixing other changes.

2. B. The record separates external claims from verification and the learner’s own reasoning.

3. C. Credentials and private environment content do not belong in an external debugging request.

4. D. The request asks for a bounded next step and preserves local verification and final explanation.

Launch round 3: notebook and workspace

1. A. Current runtime state can come from an out-of-order or no-longer-visible execution history.

2. B. A clean runtime exposes missing and out-of-order dependencies.

3. C. The interpreter exists; the evidence points to the file path supplied to it.

4. D. Those paths reveal whether installation and execution use different interpreters.

Values, Names, and Assignment

Values and assignment

1. A. Quotation marks tell Python to create a text value. The other items are a name or a symbol.

2. B. Python looks up price and quantity and multiplies them before binding the result to total.

3. C. Notebook display is an interface convenience. print() explicitly makes display part of the program.

4. D. Both calculations work, but meaningful names communicate the role of each value.

Names and changing state

1. A. Python reads the old value 8, adds 2, and rebinds score to 10.

2. B. total was calculated when quantity was 2. Earlier assignments do not rerun automatically.

3. C. Singular session_minute and plural session_minutes are distinct names.

4. D. The second line looked up 45 and bound actual to it. Reassigning planned did not rerun that line.

Trace values and bindings

1. A. The old binding, 40, is read first. Python then calculates 45 and rebinds minutes.

2. B. print(total) looks up and displays total. It does not assign a new value to total.

3. C. The role of each value matters. Morning and evening time can later change independently.

4. D. Execution order controls which values an expression reads. Calculate the result after its inputs have their intended values.

Expressions, Statements, and Execution Order

Expressions and operators

1. A. The two operands and + operator form an expression whose value is 15.

2. B. A comparison is an expression. Here it produces True even when it is not inside an if statement.

3. C. Multiplication runs before addition: 3 * 4 is 12, then 2 + 12 is 14.

4. D. A double equals sign compares. A single equals sign performs assignment.

Calls, statements, and order

1. A. The call is evaluated first and returns 8.38; assignment binds that result to rounded.

2. B. The right-side names must already have bindings when Python evaluates the expression.

3. C. Assignment stores the calculated result under subtotal for later lookup.

4. D. A clean run proves that every value is created by visible cells in the intended order.

Read a program in execution order

1. A. The parentheses group 10 + 5, so that inner expression is evaluated first.

2. B. A meaningful intermediate name makes the dependency and result easier to inspect and explain.

3. C. The inner subtraction produces -15; abs(-15) then produces 15.

4. D. Syntax can make either formula valid. The requirement determines which calculation is correct.

Python’s Words and Symbols

Tokens and identifiers

1. A. session_2 starts with a letter and contains only letters, an underscore, and a digit.

2. B. Snake case uses lowercase words separated by underscores.

3. C. Python identifiers are case-sensitive, so changing C changes the name.

4. D. 5 directly represents an integer value in the source code.

Keywords, built-ins, and operators

1. A. class has a role in Python's grammar and is reserved; course_class is an ordinary identifier.

2. B. print is a built-in function name. Rebinding it is legal but prevents normal print calls.

3. C. >= is one two-character comparison operator; no space belongs inside it.

4. D. Quotation marks create a text literal, so Python does not look up the name course.

Read Python's words and symbols

1. A. The comma delimits separate arguments inside the parentheses.

2. B. The dot separates the object expression on the left from the attribute name on the right.

3. C. After a callable expression they delimit arguments; elsewhere they can group an expression.

4. D. A comma separates adjacent arguments in a call.

Indentation and Code Blocks

Indentation with conditionals checkpoint

1. A. A colon usually introduces an indented block, such as an if body, loop body, function body, or class body.

2. B. else is a sibling of the matching if, so it should have the same indentation as that if.

3. C. Incorrect indentation can create a logic error: the program runs, but a line runs at the wrong time.

Indentation in Python checkpoint

1. D. A function body is an indented block under the def header.

2. A. The update usually belongs inside the loop so the condition can eventually become false.

3. B. A method is a function defined inside a class body, so its def line is indented under the class header.

4. C. Method definitions should usually line up with each other inside the class. If def increment(…) is inside def **init**(…), counter.increment() will not find it as a method.

5. D. Four spaces per indentation level is the standard Python style and avoids many invisible whitespace problems.

6. A. Misplaced but syntactically valid indentation can create a logic error: the code runs, but at the wrong time.

Writing Python Across Multiple Lines

Logical lines and open delimiters

1. A. Open delimiters can let one complete statement span several physical editor lines.

2. B. A newline inside an open delimiter does not finish the logical statement.

3. C. Square brackets must form a matching [and] pair.

4. D. Block indentation belongs to a colon-terminated header; continuation is controlled by open delimiters.

Readable multiline style

1. A. The previous last item already has its separator, so a new line can be added without editing that item.

2. B. Layout addresses width; meaningful intermediate names expose the calculation’s conceptual stages.

3. C. \n represents a newline character inside an ordinary string.

4. D. Implicit continuation inside matching delimiters is safer and easier to edit.

Trace code across physical lines

1. A. Newlines are allowed while the parenthesis remains open; the matching) closes the grouped expression.

2. B. Assertions or predicted outputs compare behavior rather than appearance.

3. C. Adjacent string literals are implicitly concatenated, which can hide a missing comma.

4. D. One statement per line makes the program’s sequence visible and reduces crowded edits.

Comments and Docstrings That Help

Comments that add information

1. A. The first comment records a domain rule that the arithmetic alone cannot show.

2. B. Quotation marks contain the # character, so it does not start a source comment.

3. C. Clear names make the code explain itself; comments can then cover context the names cannot.

4. D. Keeping explanation near the decision makes it easier to find and update with the code.

Comments and docstrings

1. A. Python records a string expression in the module’s first statement position as its docstring.

2. B. The docstring must be the first statement in the function body.

3. C. help(object) uses information including the object’s recorded docstring.

4. D. Comments are discarded during parsing; qualifying string statements become an object’s **doc** value.

Document code for its next reader

1. A. The first comment records the boundary between two systems and explains why conversion is necessary.

2. B. Behavior and its explanation must change together or readers receive conflicting information.

3. C. The first string statement in the function body becomes its docstring.

4. D. A narration of obvious syntax creates maintenance cost without adding knowledge.

Reading and Fixing Syntax Errors

Read an error report

1. A. Start at the final line to identify SyntaxError and read its parser clue.

2. B. The cause may be earlier, but the caret marks the parser’s detection point.

3. C. Parsing succeeds; execution fails when Python tries to look up the unknown name.

4. D. The code is valid and executable, but it represents the wrong requirement.

Common parser repairs

1. A. Double quotes can contain the apostrophe without ending the string.

2. B. Delimiter pairs must match by type: [closes with].

3. C. Two call arguments require a comma separator.

4. D. Related if and else headers are siblings and use the same indentation level.

Repair syntax methodically

1. A. An unfinished structure above often becomes impossible only when Python reaches the next statement.

2. B. A controlled edit shows whether the stated diagnosis explained the parser failure.

3. C. The parser may be unable to interpret later source reliably until the earliest failure is repaired.

4. D. Valid syntax is necessary but does not guarantee available names or correct logic.

Unit Challenge: Repair and Explain a Broken Study Planner

Broken study planner review

1. A. An underscore may occur inside an identifier; whitespace would separate two tokens.

2. B. Newlines inside the open parentheses do not finish the assignment.

3. C. Sibling branch headers align; each branch body is indented beneath its header.

4. D. The final comment records the domain reason for the calculation rather than translating visible syntax.

Values, Types, Conversion, and None

Types and behavior checkpoint

1. A. Both operands are strings, so `+` concatenates their characters.

2. B. `4.5` has type `float`, and that type’s `__name__` is the string `"float"`.

3. C. A type check identifies the value category; it does not validate the text’s meaning.

4. D. The program must explicitly choose text formatting or numeric conversion instead of relying on a guess.

Conversion boundaries checkpoint

1. A. Conversion creates a result; it does not reassign or mutate the source string.

2. B. Converting a float to int truncates toward zero, so the fractional part is discarded.

3. C. String truthiness depends on emptiness, not on interpreting the characters as a human Boolean answer.

4. D. With base 2, the digits represent four plus zero plus one, which is decimal 5.

Missing and invalid values checkpoint

1. A. `is None` directly checks for Python's missing-value marker.

2. B. The domain can assign zero a real meaning; absence is a different state.

3. C. The source type is accepted, but the string's contents cannot be interpreted as a base-ten integer.

4. D. Conversion answers whether the text can become an integer; validation answers whether that integer is allowed.

Integers and Everyday Arithmetic

Exact arithmetic checkpoint

1. A. Addition combines the two independent counts.

2. B. `abs()` returns the distance from zero, which is 12.

3. C. Parentheses combine `a` and `b` first; multiplication then doubles that result.

4. D. Python integers use arbitrary precision rather than silently wrapping at a fixed machine-word size.

Groups and leftovers checkpoint

1. A. Floor division counts complete groups and produces 4.

2. B. Remainder reports what is left after complete groups.

3. C. True division `/` preserves the fractional part of a ratio.

4. D. There are six complete groups of 60 with seven remaining.

Signed arithmetic checkpoint

1. A. Floor division chooses the lower integer below approximately -2.333, which is -3.

2. B. The result 2 satisfies `-7 == (-3 * 3) + 2`.

3. C. Parentheses make -3 the base, and multiplying -3 by -3 produces 9.

4. D. The two operations use different rules for negative fractional values.

Floating-Point Numbers and Precision

Float representation checkpoint

1. A. A measured quantity commonly accepts a nearby floating-point representation.

2. B. The arithmetic combines stored binary approximations, which need not equal the stored approximation of 0.3 exactly.

3. C. 0.125 is one eighth, and its denominator is a power of two.

4. D. The exponent notation means 1.2 multiplied by ten to the third power.

Float comparison and display checkpoint

1. A. An absolute tolerance directly states the allowed distance from zero.

2. B. An f-string always produces text; the format specification controls that text's appearance.

3. C. Formatting creates a separate string and does not modify the source float.

4. D. Tolerance is a domain decision expressed in the value's units.

Floating-point boundaries checkpoint

1. A. `math.isfinite()` rejects both positive/negative infinity and NaN.

2. B. NaN is intentionally unequal to every value, including itself; `math.isnan()` is explicit.

3. C. Python's built-in float division raises rather than silently producing infinity.

4. D. Small representation and rounding effects can build across repeated operations.

Money, Exact Ratios, and Complex Numbers

Decimal arithmetic checkpoint

1. A. Constructing directly from text avoids creating a binary float approximation first.

2. B. The quantum controls the decimal exponent, and the named rounding mode controls halfway behavior.

3. C. Both stages are computable; the business rule determines which result is meaningful.

4. D. The program must choose a representation rather than silently merging two precision models.

Exact ratios checkpoint

1. A. Fraction reduces the numerator and denominator by their common factor.

2. B. One third is two sixths; adding one sixth gives three sixths, normalized to one half.

3. C. The constructor receives the stored float value, not the original source characters.

4. D. The maximum denominator constrains how simple the approximating ratio may be.

Complex-number checkpoint

1. A. Python uses `j` for the imaginary unit in a complex literal.

2. B. The magnitude is the square root of 3 squared plus 4 squared, which is 5.

3. C. The conjugate retains the real component and reverses the imaginary sign.

4. D. Magnitude supplies the domain-specific ordering that direct complex less-than does not define.

Strings and Unicode Text

String boundaries checkpoint

1. A. The first sequence element is zero positions from the beginning.

2. B. The slice includes indexes 1, 2, and 3, but excludes stop index 4.

3. C. Negative indexes count backward from the end; -1 is the last element.

4. D. A slice clips at the available boundary instead of requiring every index to exist.

String cleanup checkpoint

1. A. Strings are immutable; `strip()` returns a separate value.

2. B. `removeprefix()` states an exact beginning marker; `strip()` treats its argument as individual edge characters.

3. C. An immutable string operation produces a result that must be stored if it is needed later.

4. D. Casefolding creates a comparison-oriented form across a wider range of Unicode case behavior.

Searching and Unicode checkpoint

1. A. Membership directly produces the Boolean fact the question asks for.

2. B. `find()` uses -1 as its not-found sentinel.

3. C. A composed character and a base-plus-combining sequence can render alike while containing different code points.

4. D. Joining text does not guess how non-string values should be converted or formatted.

Input, Output, and Clear Formatting

Console input and output checkpoint

1. A. Console input returns the typed characters as a string.

2. B. `sep` is placed between the three displayed arguments.

3. C. The raw input is text; explicit conversion gives later arithmetic a numeric operand.

4. D. An empty end string suppresses the usual trailing newline for that call.

Clear formatting checkpoint

1. A. The colon begins the specification and `.2f` requests two fixed-point decimal places.

2. B. Percentage formatting scales by 100 and shows one decimal place.

3. C. The greater-than marker points toward the right edge of the field.

4. D. F-strings produce text and do not modify the source value.

Input-to-output checkpoint

1. A. Distinct names let a reader inspect the source text and the converted value independently.

2. B. `!r` requests the diagnostic `repr()` representation.

3. C. Keep numeric values for arithmetic and create text only for presentation.

4. D. A separate failure experiment preserves useful evidence without introducing an unexplained recovery mechanism.

Booleans, Comparisons, and Truth

Comparison facts checkpoint

1. A. Greater-than-or-equal includes the boundary value 18.

2. B. The chained comparison states both inclusive boundaries around score.

3. C. The operands remain text, so lexicographic character order applies.

4. D. `math.isclose()` expresses relative and/or absolute tolerance rather than exact stored equality.

Truthiness checkpoint

1. A. Numeric zero is falsy in a Boolean context.

2. B. Truthiness checks string emptiness, and one space is one character.

3. C. `bool()` does not parse human Boolean words; it checks whether the string is empty.

4. D. Identity with None preserves zero as a present value.

Boolean operators checkpoint

1. A. `not` reverses the truth value.

2. B. `and` requires both Boolean conditions to hold.

3. C. The first operand is falsy, so `or` evaluates and returns the second operand.

4. D. `and` short-circuits as soon as it finds a falsy operand.

Dates, Times, and Durations

Date and time types checkpoint

1. A. A `date` contains year, month, and day.

2. B. A timedelta stores a duration that can be added to or subtracted from calendar values.

3. C. `date.fromisoformat()` validates and constructs a date from supported ISO text.

4. D. The date constructor validates the relationship among year, month, and day.

Duration and parsing checkpoint

1. A. Date subtraction produces the elapsed duration between the calendar days.

2. B. `total_seconds()` converts the complete duration, while `.seconds` is only the normalized leftover component.

3. C. Parsing succeeds only when the source characters match the declared format contract.

4. D. Thirty elapsed days is precise, but a calendar-month requirement needs separate rules.

Timezone meaning checkpoint

1. A. The +00:00 offset makes the datetime aware and identifies its relationship to UTC.

2. B. `astimezone()` preserves the instant while calculating the target zone's wall-clock representation.

3. C. A naive datetime has calendar and clock fields but no unambiguous global-time relationship.

4. D. A changing clock makes exact expected values unstable.

Unit Challenge: Build a Date-Aware Purchase Summary

Chrononaut supply clearance checkpoint

1. A. Constructing from source text avoids carrying an earlier binary-float approximation into Decimal arithmetic.

2. B. The supplied absolute tolerance expresses the acceptable measurement difference.

3. C. A non-None tzinfo indicates an aware datetime with offset or timezone information.

4. D. Separating calculation from presentation preserves useful types until the artifact is ready to display.

Milestone Challenge: Pack a Moon-Garden Supply Box

Check the reasoning behind your result

1. A. Integer cents represent whole-cent prices exactly. Display formatting, input validation, and syntax are separate concerns.

Lists: Keeping Values in Order

Positions and slices checkpoint

1. A. Index -1 selects the final item, which is the string "torch".

2. B. A slice includes its start and excludes its stop.

3. C. Slice bounds are clipped to the available range; they do not demand an exact final index.

4. D. A full slice with step -1 produces a new list in reverse traversal order.

Mutation and return values checkpoint

1. A. append() adds its argument as one new item, even when that argument is another list.

2. B. pop(index) mutates the list and returns the removed item.

3. C. remove(value) searches from the beginning and deletes only the first equal value.

4. D. In-place list methods normally return None; the mutation is already visible through the original list.

Aliases and copies checkpoint

1. A. List assignment creates another reference to the same object; it does not copy the list.

2. B. copy() creates a new outer list populated with references to the current items.

3. C. A shallow copy separates the outer list but retains references to its existing item objects.

4. D. The loop target receives each value; rebinding that name does not mutate the container position.

Tuples and Unpacking Fixed Records

Tuple syntax and immutability checkpoint

1. A. The comma creates the one-item tuple; parentheses alone only group the integer expression.

2. B. Slicing a tuple returns a new tuple containing the selected positions.

3. C. A tuple’s positions cannot be replaced in place.

4. D. Tuple immutability protects its positions, not the internal state of mutable objects referenced there.

Unpacking shapes checkpoint

1. A. The first supplied value is assigned to the first target name.

2. B. Without a starred target, the number of target names must match the supplied value count exactly.

3. C. A starred assignment target collects its values into a new list.

4. D. The two values fill the ordinary targets, leaving an empty list for the starred target.

Fixed-record decisions checkpoint

1. A. A two-item tuple preserves the coordinate’s fixed positional order.

2. B. The route’s ordered structure is intended to grow and shrink, which matches a list.

3. C. The string and integer in ("zone", 3) are hashable; the other tuples contain lists or a dictionary.

4. D. Ordinary unpacking requires one target per supplied value; the error exposes a shape mismatch.

Dictionaries: Finding Values by Key

Dictionary lookup checkpoint

1. A. Direct dictionary membership tests keys.

2. B. Square-bracket lookup requires the key and raises KeyError when it is absent.

3. C. get() returns the supplied default for absence and does not mutate the dictionary.

4. D. Membership reports whether the key exists even when the associated value is None.

Updates, removals, and views checkpoint

1. A. For duplicate keys, the right mapping’s value wins in a merge.

2. B. Dictionary pop removes a specified key and returns its associated value.

3. C. Dictionary views are dynamic windows onto the current mapping.

4. D. Direct dictionary traversal supplies keys in insertion order.

Dictionary modeling checkpoint

1. A. A string field key provides direct named lookup in the record mapping.

2. B. A counter maps each distinct value to the number of times it occurred.

3. C. A grouping dictionary uses the category as key and a collection of matching items as its value.

4. D. Dictionary copy is shallow: it separates the outer mapping while retaining current value references.

Sets: Uniqueness and Group Comparisons

Set creation and mutation checkpoint

1. A. Empty braces create a dictionary; set() is the empty-set constructor.

2. B. discard() removes a member if present and treats absence as a no-op.

3. C. Sets have no last positional member; pop removes an arbitrary member and returns it.

4. D. A set retains one member for each distinct equal value.

Set operations and relationships checkpoint

1. A. Intersection uses & and keeps members shared by both sets.

2. B. Set difference keeps members on the left that are absent from the right.

3. C. required <= available asks whether required is a subset of available.

4. D. isdisjoint() is true exactly when the sets have no members in common.

Hashability and frozenset checkpoint

1. A. A tuple of hashable integers is hashable; the list, dictionary, and mutable set are not.

2. B. frozenset offers immutable set membership and non-mutating operations.

3. C. Frozen sets are hashable when their members are hashable, so they can be members of another set.

4. D. The two frozen sets have equal members and therefore represent the same dictionary key.

Visiting Items in a Collection

Collection traversal checkpoint

1. A. Direct list traversal supplies list items in their sequence order.

2. B. The items view supplies a two-item tuple for each mapping pair.

3. C. An empty list supplies no items, so the body does not execute.

4. D. Set traversal covers current membership without establishing a sequence-position contract.

Range and enumerate checkpoint

1. A. range(stop) starts at zero and excludes the stop.

2. B. A negative step descends from 5 and the excluded stop 0 leaves 1 as the final value.

3. C. enumerate supplies (index, value) tuples, commonly unpacked into two names.

4. D. enumerate still visits every item; start controls only the first supplied counter.

Alignment and mutation checkpoint

1. A. zip groups the next value from each iterable by traversal position.

2. B. zip(..., strict=True) raises ValueError if an input ends before the others.

3. C. Local records preserve each relationship even when records are inserted, removed, or moved.

4. D. Structural mutation changes the position layout being traversed, so the next logical item can be skipped.

Sorting Collections Without Losing Meaning

Sorting state checkpoint

1. A. sorted() returns a new list from any sortable iterable and leaves the input collection unchanged.

2. B. list.sort() changes the list in place and returns None.

3. C. reverse() flips existing order; it does not calculate ascending or descending value rank.

4. D. The sorted() built-in always returns a list.

Sort-key checkpoint

1. A. The key computes a casefolded comparison value while the result retains each original string.

2. B. itemgetter("score") returns a callable that retrieves the score from each supplied record.

3. C. The values are legal list members but do not share a defined ordering relation.

4. D. A string key orders character sequences, which is a different rule from numeric magnitude.

Stable record sorting checkpoint

1. A. A stable sort does not reverse or shuffle the earlier relative order of equal-key items.

2. B. Moving complete records preserves the meaning of related fields.

3. C. Apply the secondary time rule first, then the primary score rule; stability retains time order within score ties.

4. D. Items supply complete key-value tuples, which remain connected during sorting.

Building and Reading Nested Data

Nested-path checkpoint

1. A. Index the outer list, then use dictionary keys for the weather record and its temperature field.

2. B. The coordinate field stores the fixed tuple (9, 2).

3. C. The operation is applied to the outer list before Python can inspect any record IDs.

4. D. Square-bracket lookup of an absent dictionary key raises KeyError.

Nested-shape boundary checkpoint

1. A. Each integer index applies to the list produced by the previous level.

2. B. Membership distinguishes an existing key from absence before the stored value is checked.

3. C. Lookup returns the referenced nested value; assignment to another name does not copy it.

4. D. The unpacking ValueError exposes a fixed-shape mismatch.

Nested collection choice checkpoint

1. A. The list preserves record order while each dictionary names its fields.

2. B. A keyed lookup dictionary connects the stable vault ID to its record.

3. C. Lengths can match even after one list has been reordered or replaced incorrectly, leaving fields semantically misaligned.

4. D. Multiple deliberate views can serve distinct access patterns while one documented source remains authoritative.

Unit Challenge: Decode the Starship Signal Vault

Signal-vault collection decisions

1. A. A dictionary preserves the direct key-to-record relationship.

2. B. Set update consumes the record’s symbol members and preserves unique membership.

3. C. The source order is evidence, so ranking must be a separate list.

4. D. Sorting the complete (position, letter) clues puts letters into their encoded code positions.

Choosing a Path with `if`, `elif`, and `else`

Branch structure checkpoint

1. A. Alignment with the if header places the statement after the conditional block.

2. B. else is the fallback for the chain; it has no separate condition.

3. C. A skipped assignment does not bind the name. A complete fallback or earlier default prevents that path.

4. D. The colon ends the compound-statement header and introduces its indented suite.

Branch choice checkpoint

1. A. A branch chain chooses the first true branch and then continues after the complete chain.

2. B. Independent if statements allow zero, one, or both bodies to run.

3. C. The broader first condition captures all values the later condition could accept.

4. D. Both less-than comparisons are false at 60, so the fallback runs.

Rule shape checkpoint

1. A. Membership tests whether the command equals one of the set members.

2. B. Identity with None preserves the distinction between missing and valid falsy values such as 0.

3. C. Nesting can represent a dependent question and preserve a different outcome for each failed stage.

4. D. A conditional expression is clearest for one compact two-way value choice.

Combining Conditions Without Surprises

Boolean policy checkpoint

1. A. Both requirements must be true, so and is the appropriate connective.

2. B. Parentheses group the two visitor alternatives, while and makes the open room mandatory.

3. C. Inclusive or is false only when neither operand is truthy.

4. D. The two alternatives explicitly allow A alone or B alone, excluding both and neither.

Short-circuit checkpoint

1. A. A false left operand determines the and result, so the unsafe index is not evaluated.

2. B. Membership is evaluated first and protects the lookup when the key is absent.

3. C. A truthy left operand determines the or result, so Python does not evaluate the right side.

4. D. Truthiness collapses zero with other falsy values even when the program gives zero a distinct meaning.

Complete rule checkpoint

1. A. Negating a conjunction negates each fact and changes and to or.

2. B. The two required ranges do not overlap, so no number satisfies both.

3. C. With no falsy item to disprove the claim, all() returns True for an empty iterable.

4. D. The first operand requires non-emptiness; only then does all() verify every supplied fact.

Turning Rules into Complete Decisions

Complete cases checkpoint

1. A. A branch chain chooses at most one category, and else can cover every remaining expected case.

2. B. Independent checks allow both notices to be collected when both facts apply.

3. C. Strict comparisons on both sides leave an uncovered boundary at 10.

4. D. Just below, exactly at, and just above expose endpoint mistakes.

Precedence and evidence checkpoint

1. A. The highest-priority overlapping rule must be checked before lower-priority pricing rules.

2. B. Distinct values preserve distinct meanings that truthiness alone would collapse.

3. C. The reason exposes the path and policy interpretation that produced the result.

4. D. Two selected rows mean the written cases overlap for that sample.

Discrete choices checkpoint

1. A. A menu of discrete literal commands maps naturally to cases.

2. B. The vertical bar joins alternative patterns that share one body.

3. C. The wildcard matches any value not handled by an earlier case.

4. D. Comparisons in an ordered branch chain directly express numeric ranges.

Building Results with `for` Loops

Accumulator foundations checkpoint

1. A. An empty list represents an ordered result before any accepted item has been appended.

2. B. Initialization inside the body resets accumulated state and can leave no result for empty input.

3. C. Zero is a valid reading: its numeric contribution is zero, while it is still one accepted item.

4. D. The two situations have different meaning and should not be collapsed into the same value.

Transform and filter checkpoint

1. A. A direct transformation maps each source value to one result value.

2. B. The acceptance condition controls whether that item is added.

3. C. A rejection collection makes loss observable and supports debugging or reporting.

4. D. The condition ensures the conversion executes only on inputs this simple policy accepts.

Structured result checkpoint

1. A. get returns the default zero for an absent key, then the expression stores one.

2. B. Each group needs a destination list that can receive all records sharing that key.

3. C. The set detects repeats while the list preserves the first occurrence order.

4. D. An empty iterable provides no candidate for a minimum.

Repeating Until Something Changes

While-loop progress checkpoint

1. A. while is a pre-test loop; a false initial condition skips the body entirely.

2. B. The body runs for positive values; the check at zero is false.

3. C. The body visits 0, 3, 6, and 9, then updates to 12 before the next false check.

4. D. Without guaranteed progress or another bounded exit, the continuation condition may remain true forever.

Bounded stopping checkpoint

1. A. A retry budget is a product requirement when unlimited attempts would be unsafe or unhelpful.

2. B. A sentinel is recognized and handled as control information, so it should not enter normal results.

3. C. The final condition alone may not communicate why processing ended.

4. D. Recognizing the stop signal before normal processing prevents it from being treated as data.

Safe while-loop design checkpoint

1. A. Every iteration path must make progress before it can skip the remaining body.

2. B. Direct traversal expresses collection-driven repetition without manual index state.

3. C. Multiple stop facts can coincide; the reporting policy should deliberately choose the main reason.

4. D. A before/after state trace exposes skipped updates, overshoot, and the exact stopping transition.

Searching, Skipping, and Stopping Loops

Search contract checkpoint

1. A. Once the first match is found, later evidence cannot improve that contract.

2. B. A later match must be allowed to replace an earlier one.

3. C. continue affects only the current iteration and leaves the loop active.

4. D. An initialized list remains empty when no item satisfies the condition.

Loop exit checkpoint

1. A. Loop else represents normal completion rather than a particular if condition.

2. B. The loop exhausts normally immediately, so its else suite runs.

3. C. The state distinguishes break causes that loop else alone cannot name.

4. D. Both for and while skip their else suite when terminated by break.

Control choice checkpoint

1. A. break applies to the innermost enclosing loop.

2. B. pass is a no-operation placeholder, unlike continue.

3. C. Membership states the complete yes/no question directly.

4. D. A loop can preserve richer artifacts than a direct Boolean membership result.

Working Through Grids and Pairs

Grid traversal checkpoint

1. A. The outer collection contains rows, so each outer iteration supplies one row.

2. B. A fresh inner traversal begins for each row.

3. C. Direct traversal does not request positions that the current row lacks.

4. D. The first row’s width is not proof that every later row is equally wide.

Pair generation checkpoint

1. A. Each of two outer values combines with all three inner values: 2 × 3.

2. B. zip advances the inputs together rather than forming all combinations.

3. C. Only later positions remain available for the right side.

4. D. The inner loop is fully consumed for each outer item in its supplied order.

Nested exits and work checkpoint

1. A. break exits only the nearest enclosing loop.

2. B. The outer loop needs its own explicit decision to stop.

3. C. Normal inner exhaustion triggers that loop’s else suite.

4. D. Preparing invariant data once avoids rebuilding the same value for every combination.

Building Collections with Comprehensions

List comprehension checkpoint

1. A. The leading expression produces one collected result for each supplied value.

2. B. The trailing if is a filter controlling whether the current item contributes output.

3. C. A failing trailing condition prevents evaluation of the output expression for that item.

4. D. The comprehension creates a new collection and does not assign through source positions.

Comprehension shape checkpoint

1. A. The conditional expression chooses one of two outputs for each supplied item.

2. B. Set semantics retain unique equal results.

3. C. Dictionary comprehensions construct a key-value pair for each accepted input.

4. D. The comprehension’s set shape is preserved even when it collects no values.

Comprehension boundaries checkpoint

1. A. Dictionary assignment semantics apply; a repeated equal key receives the latest value.

2. B. Comprehension for clauses follow the same left-to-right nesting order as explicit loops.

3. C. The purpose is an action, not a collected result, so an explicit loop communicates it.

4. D. Several artifacts and early control flow need the space and statements of an explicit loop.

Unit Challenge: Escape the Clockwork Maze

Clockwork-maze control decisions

1. A. Progress before continue prevents the loop from processing the same command forever.

2. B. A falsy row check stops and before Python indexes the maze with an invalid row.

3. C. Each collection answers a different route question without sacrificing the other relationship.

4. D. Successful escape fulfills the contract, so further tape processing is unnecessary.

Defining Functions and Returning Useful Results

Definition and call checkpoint

1. A. The body is recorded as behavior and runs only when a call later requests it.

2. B. Parentheses after the function value perform a call with the enclosed argument.

3. C. Definitions name parameters; calls supply argument values for those parameters.

4. D. No parentheses appear, so assignment stores a reference to the callable itself.

Return value checkpoint

1. A. Return completes that frame and makes the call expression evaluate to the returned value.

2. B. Displayed output is not returned data; reaching the end returns None.

3. C. The explicit sentinel makes the no-match contract visible even though falling off the end would also return None.

4. D. A return value crosses the call boundary and remains available to caller code.

Return paths and call order checkpoint

1. A. Return immediately completes the current call.

2. B. The middle path reaches the end and therefore returns None.

3. C. The comma constructs a tuple that the caller may unpack.

4. D. Python must evaluate the inner call to obtain the outer call’s argument.

Designing Clear Parameters and Calls

Argument binding checkpoint

1. A. Argument expressions must be evaluated before Python can bind parameters and enter the called body.

2. B. Explicit keyword labels bind values by parameter name.

3. C. Python rejects multiple values for one parameter before entering the body.

4. D. Positional values bind from left to right to positional parameters.

Defaults and keyword-only checkpoint

1. A. Omission selects the default; explicitly supplied falsy values remain real arguments.

2. B. Later omitted arguments bind to the same default object.

3. C. Per-call initialization occurs inside the body only when no list was supplied.

4. D. Parameters after the bare star are keyword-only.

Variadic and unpacking checkpoint

1. A. The variadic positional parameter collects supplied positional values into a tuple.

2. B. Double-star collection creates a dictionary from extra named arguments.

3. C. A star at the call expands iterable items into positional arguments.

4. D. Double-star expansion turns mapping entries into keyword arguments governed by normal binding rules.

Breaking a Program into Small Functions

Function responsibility checkpoint

1. A. A useful contract states important edge behavior such as the result for zero.

2. B. Calculation rules and presentation rules can change independently and produce different reusable results.

3. C. Mutation changes state supplied by the caller and is therefore part of the function contract.

4. D. An explicit parameter exposes a changing dependency at the call boundary.

Data flow and side effects checkpoint

1. B. Printing is an effect; reaching the end without return produces `None`.

2. C. A pure result depends on inputs and does not accumulate hidden changing state.

3. D. A command is called for its effect; the name and contract should reveal that effect.

4. A. Named stages make the direction and meaning of returned data visible.

Composed report checkpoint

1. C. Each stage returns the kind of value needed by the next stage.

2. D. One validation boundary prevents later stages from repeating the same rule.

3. A. The empty-input policy still returns a useful complete report.

4. B. Source preservation is an observable part of this pipeline’s contract.

Following Names Through Function Calls

Call frame checkpoint

1. A. The same spelling can refer to separate local names owned by separate call frames.

2. B. The suspended caller retains its frame and return point.

3. C. The returned nested function closes over `prefix` from its enclosing function scope.

4. D. A parameter makes a changing dependency visible and independently configurable.

Name assignment checkpoint

1. B. Nearest-scope lookup finds the integer binding before the built-in function.

2. C. An ordinary assignment classifies that name as local throughout the function body.

3. D. Shared changing state creates coupling between calls and other code.

4. A. The closure keeps access to the specific enclosing binding used by that factory call.

Closure state checkpoint

1. B. `nonlocal` targets the nearest enclosing function binding with that name.

2. C. Separate factory calls create separate captured state.

3. D. The shared name is resolved later when each closure is called.

4. A. The caller frame and its locals remain while the callee runs.

Passing Functions as Values

Function value checkpoint

1. A. Without call parentheses, the expression refers to the callable object.

2. B. The sorting mechanism invokes the supplied key callback with individual records.

3. C. The expression calls immediately and supplies its result where `sorted` expects a callable.

4. D. The mechanism calls the key function with each list item.

Callback contract checkpoint

1. B. The selector uses each predicate result as its keep-or-skip decision.

2. C. A transform callback produces the output representation for each input item.

3. D. A stable shared contract means selection can vary while the call shape remains the same.

4. A. Behavior-as-data should solve a real configuration or reuse need rather than add ceremony.

Configurable behavior checkpoint

1. A. The factory call creates and returns configured predicate behavior.

2. B. A short local expression can remain readable as a lambda.

3. C. The tuple gives the sorting mechanism a primary and secondary key.

4. D. Equal key values retain the original relative order.

Solving Smaller Problems with Recursion

Recursive progress checkpoint

1. A. The explicit smallest case returns directly and stops adding frames.

2. B. The recursive argument is measurably smaller on every call.

3. C. The caller suspends its addition until the child call returns a value.

4. D. The newest active call returns first, allowing its caller to resume.

Nested tree checkpoint

1. B. One leaf contributes one to its parent’s count.

2. C. The parent combines independent results returned by smaller recursive calls.

3. D. The sentinel reports that the complete subtree was searched without a match.

4. A. Each returning parent adds its own contribution to the child result.

Recursive fit checkpoint

1. A. Flat sequential repetition maps directly to a loop and one frame.

2. B. The data's own definition repeats at smaller child nodes.

3. C. The depth failure is evidence to inspect the base case and recursive argument.

4. D. Delegation relays each value produced while traversing the smaller child tree.

Producing Values One at a Time

Iterator state checkpoint

1. A. The iterable supplies an iterator that will yield its items in sequence.

2. B. Consumption state belongs to that iterator, not to every possible iterator over the collection.

3. C. The default form returns the fallback instead of exposing `StopIteration`.

4. D. Each iterator tracks consumption independently even though both read the same list.

Generator timing checkpoint

1. B. Calling a generator function prepares suspended iteration; the first request starts the body.

2. C. The generator preserves both its frame and the exact resume position.

3. D. Resume continues after the previous yield with local state intact.

4. A. Return finishes the generator; yielded values are the ordinary stream items.

Lazy pipeline checkpoint

1. C. The generator expression continues from its current one-pass position.

2. D. The consumer requests only as much input as it needs to determine its answer.

3. A. A new generator-function call creates a new generator object and frame.

4. B. Delegation forwards the sub-iterable's values through the outer generator.

Adding Behavior with Decorators

Decorator transformation checkpoint

1. A. Decorator syntax passes the newly defined function to the decorator and rebinds its name to the result.

2. B. The decorator installs the replacement at definition time.

3. C. Returning the wrapper and executing the wrapper are separate events.

4. D. The public call returns what the wrapper returns, so an omitted return loses the original value.

Wrapper contract checkpoint

1. B. Expansion forwards both collected positional and keyword arguments.

2. C. `wraps` updates the wrapper with key metadata from the original callable.

3. D. Forwarding and returning protect both sides of the original call contract.

4. A. The wrapper retains access to state created when the decorator received the function.

Configured decorator checkpoint

1. B. The configuration layer returns the function-transforming decorator layer.

2. C. Recording afterward provides the real result and avoids recording completion before it occurs.

3. D. Stacked decorators apply from the bottom upward, like nested function calls.

4. A. Call results return outward through the nested wrappers.

Unit Challenge: Unlock the Moonlit Library

Moonlit Library evidence check

1. D. Generator objects own one-pass state; a new call creates a complete traversal from the root.

2. A. A scroll is a leaf in this archive shape and needs no recursive child call.

3. B. Traversal yields every scroll; the supplied predicate owns eligibility policy.

4. C. The decorator adds an effect while preserving the public function’s returned result.

When Names Share the Same Object

Bindings and shared objects checkpoint

1. A. Ordinary assignment adds or changes a binding; it does not clone the object on the right.

2. B. Mutation changes the shared object, so every remaining reference observes its new state.

3. C. The assignment redirects one name and leaves the object reached by other names unchanged.

4. D. Value equality does not require object identity.

Equality and identity checkpoint

1. B. Equality compares list contents; identity asks whether there is one list object.

2. C. None is a singleton sentinel and is checked by identity.

3. D. An ID is useful as temporary same-object evidence, not as a persistent public identifier.

4. A. Interning can make identity experiments misleading; == expresses the value contract.

Party roster checkpoint

1. C. Outer records can be independent while a particular nested path remains shared.

2. D. The function copied the starting values into a new player-owned list.

3. A. The field uses the None singleton as its absence sentinel.

4. B. The command’s documented effect is the useful behavior; its result is deliberately None.

Changing Objects and Choosing Safe Keys

Mutation and rebinding checkpoint

1. A. In-place mutation changes the existing object’s contents without replacing that object.

2. B. A string object cannot change its contents; its transformation produces another string.

3. C. The assignment stores the method’s result, not the list that the method changed.

4. D. List addition returns a new list; assignment then changes one binding.

Type-dependent change checkpoint

1. B. List in-place addition changes the existing list, so its aliases observe the new element.

2. C. The immutable tuple does not change; augmented assignment binds the target to the returned tuple.

3. D. Container immutability does not recursively freeze mutable descendants.

4. A. The same syntax can use different behaviors supplied by different types.

Expedition state checkpoint

1. A. The complete coordinate value is hashable and stable for set membership.

2. B. The in-place command changes state reached through both aliases.

3. C. Copy depth follows the fields the function promises to change independently.

4. D. Set members require hashable values whose lookup behavior remains stable.

Copying Nested Data Without Surprises

Shallow object graph checkpoint

1. A. List repetition repeated the row reference rather than constructing three row objects.

2. B. Each evaluation of the list literal creates a distinct row object.

3. C. A shallow copy duplicates the outer container and reuses contained references.

4. D. Outer independence does not separate a reused inner reference.

Selective ownership checkpoint

1. B. Sharing is safe when no allowed operation can change the shared value in place.

2. C. A distinct notes dictionary would still leak appends if its nested entries list remained shared.

3. D. Ownership and allowed mutation, not visual nesting alone, determine independence.

4. A. Selective copying makes ownership decisions explicit as the schema evolves.

Mission board checkpoint

1. B. Both mutable levels are inside the new board’s ownership contract.

2. C. The observable nested source remains unchanged because its list is independently owned.

3. D. A shallow copy separates the outside and reuses the nested task reference.

4. A. Deep copying is justified by whole-graph independence, not by a preference for maximum depth.

Designing Functions That Control Mutation

Call boundary checkpoint

1. A. The local parameter binding receives a reference to the evaluated argument object.

2. B. Caller and callee names occupy different namespaces even when they initially reach one object.

3. C. Both caller and parameter references reach the list whose state was changed.

4. D. The public name, docstring, effect, and result all agree.

Mutation contract checkpoint

1. B. sorted returns a list; list.sort changes the list and returns None.

2. C. Returning the input after mutation exposes the same object under another name.

3. D. The one default list can therefore be reused by multiple omitted-argument calls.

4. A. Construction moves from definition time to the relevant call time.

Quest API checkpoint

1. A. The mutation targets function-created state rather than the caller’s source graph.

2. B. Value and identity evidence together show per-call ownership.

3. C. The immutable sequence representation prevents an output alias to internal mutable lists.

4. D. The docstring makes explicit-input ownership different from fresh omitted-input state.

Unit Challenge: Break the Mirror Vault

Mirror Vault evidence check

1. D. A shallow outer copy reuses the nested journal reference.

2. A. The complete tuple is hashable and equal tuple values can find the same set member.

3. B. Mutation targets the newly owned inventory rather than the supplied explorer graph.

4. C. The immutable sequences prevent the result from being a mutable alias to journal internals.

Milestone Challenge: Route Supplies through a Puzzle Village

Check the reasoning behind your result

1. B. Assignment creates another reference to the same dictionary; copying separates these mappings of immutable integers.

Turn a Vague Idea into Testable Examples

Rules and observable behavior

1. B. Travel time is consulted after darkness ties and before input order.

2. C. A running-best loop and a non-mutating sorted approach could both preserve the stated behavior.

3. B. The contract promises one name, not a record, list, or printed side effect.

Boundaries and acceptance evidence

1. C. Values just below, exactly at, and just above 7 expose the inclusive threshold.

2. A. An implementation may handle an empty list correctly while forgetting to exclude unavailable records.

3. D. The hand-calculated value does not repeat the implementation being checked.

Is the specification ready?

1. A. The damaged pod would win the score comparison unless eligibility is applied first.

2. B. A supported tie can otherwise produce an arbitrary or unstable choice.

3. D. Input scale and order preservation can be checked and can influence an algorithm choice.

Turn a Plan into Working Python

From manual work to function boundaries

1. D. Each usable candidate is compared with the best candidate found so far.

2. A. Eligibility is a domain decision with a clear Boolean result.

3. C. Keeping the existing candidate preserves the earlier-input tie rule.

Build and check one behavior at a time

1. B. The slice is small but complete from public input to observable result.

2. D. The earliest incorrect stage is the narrowest useful place to inspect.

3. A. sorted returns a new list, while the listed methods mutate the existing list.

Carry a plan into working Python

1. C. The changed behavior must be explicit and observable before implementation is edited.

2. A. Separating calculation from display gives each responsibility a direct interface.

3. B. Without the update condition, the plan does not explain how later candidates can change the result.

Choose an Algorithm That Fits

Choose a single-traversal pattern

1. C. Every input produces one output in the same position.

2. B. The counter records how many matches exist in the processed prefix.

3. D. The contract is satisfied at the first qualifying encounter.

Choose state for selection and mappings

1. B. Equality does not satisfy >, so the existing first-seen best remains.

2. C. Grouping retains several values per key, so each tower maps to a collection.

3. A. The two structures protect two different promises: efficient membership and ordered output.

Adapt and combine patterns

1. D. The comparison uses values[index - 1], so the first valid pair ends at index 1.

2. A. The contract returns the end position as a processed-symbol count, while the slice excludes index 5.

3. B. Clear named stages can be worth their intermediate data when they communicate or support real reuse.

Know It Works and Understand the Cost

Build a correctness claim

1. B. The repeated oak is not adjacent to its first occurrence, so neighbor comparison misses it.

2. D. The algorithm’s seen set can store only hashable values.

3. A. At completion, the statement about all processed values applies to every input value.

Reason about growing inputs

1. C. One check per item gives a count proportional to n.

2. A. The algorithm must inspect every item when no earlier match ends the search.

3. D. Big-O describes growth, not an exact wall-clock duration.

Choose a defensible trade-off

1. D. The operation inside the loop repeats comparisons against more earlier values.

2. A. Hash-table membership improves ordinary lookup growth while requiring stored hashable keys.

3. C. Returning sorted distinct values violates a contract that preserves encounter order.

Unit Challenge: Wake the Clockwork Observatory

Clockwork observatory calibration check

1. C. Not replacing on equality implements the first-appearance tie rule.

2. A. The contract returns the 1-based end, equal to the exclusive slice end and processed-symbol count.

3. D. The complete calibration cannot finish, so the contract returns None.

4. B. The returned step must include the reading that first reaches the target.

5. C. Up to about n candidates each require work tied to the window width.

Understand Failures and Read Tracebacks

Choose the first useful evidence

1. B. The parser cannot form the statement, so inspect syntax before ordinary execution or exception handling.

2. D. A logic bug needs an oracle. The independent expected value makes the silent mismatch observable.

3. A. The final line identifies what stopped execution; frames then explain where and through which calls.

Use exception names as clues

1. B. The mapping's keys show the actual shape, while the contract decides whether absence is normal or defective.

2. D. `int` accepts strings, but `many` is not a valid integer representation.

3. A. A traceback localizes the failed operation; data flow may lead to an earlier source of the broken assumption.

Turn a failure into an investigation

1. C. Python checks language operations, not the unstated festival rule. An oracle makes the mismatch observable.

2. A. Representation and type directly test the hypothesis while preserving the failing value.

3. D. A clean top-to-bottom run recreates definitions and state from the saved source.

Raise Useful Errors and Handle Expected Failures

Separate normal results from contract failures

1. D. No match can be an ordinary search outcome, and a documented `None` makes that branch explicit.

2. D. The integer type is acceptable, but zero violates the supported value range.

3. C. The message supplies the location and broken rule without dumping private or irrelevant state.

Protect only the operation you understand

1. B. A narrow block prevents unrelated programming defects from being mislabeled as invalid input.

2. A. `else` separates the successful continuation from the operations covered by exception handlers.

3. C. Cleanup should run without erasing the original result or failure evidence.

Keep context without losing the cause

1. B. Chaining combines domain location with the original technical reason.

2. C. Public validation should not depend on assertion execution and should communicate type or value violations directly.

3. D. The boundary only knows how to handle anticipated command failures; an unrelated defect requires investigation.

Debug One Hypothesis at a Time

Turn a report into stable evidence

1. D. A frozen reproduction lets you tell whether a later change affects the original failure.

2. C. A minimal case reduces competing explanations and makes the first wrong transition easier to see.

3. D. It identifies a mechanism and predicts an observation that could prove it wrong.

Find the first divergence

1. B. A passing intermediate boundary localizes this case without proving unrelated cases.

2. A. Those fields expose the relevant transition and allow expected-versus-actual comparison.

3. C. A causal hypothesis should predict observable changes in the failure and nearby cases.

Prove the repair survives

1. B. The minimal reproduction directly guards the behavior that failed and is easy to understand.

2. A. A root repair corrects the earliest wrong decision and remains consistent across related inputs.

3. C. Only controlled runs against the actual contract and environment can support or reject the suggestion.

Pause, Inspect, and Shrink a Bug

Know what the debugger has and has not executed

1. A. A line breakpoint pauses before the highlighted operation, so stepping over reveals its after-state.

2. C. Step into enters the called function so its parameters and internal transitions become visible.

3. B. Each frame has its own local state; moving between frames shows values passed across boundaries.

Choose evidence over the initial story

1. B. An oracle is part of the investigation and must also withstand independent checking.

2. A. A side-effect-free condition pauses only when the relevant iteration reaches the chosen line.

3. D. Temporary interactive stops should not surprise later users or automated runs.

Build a runnable bug report

1. C. Useful minimal examples remove noise while retaining everything required to run and observe the same behavior.

2. A. One-at-a-time reductions reveal which details are causally required and which are noise.

3. D. Sanitization can alter behavior, so the safe version must be verified as a faithful reproduction.

Unit Challenge: Repair the Glitched Arcade

Glitched arcade repair check

1. B. The contextual error tells where the command failed, while its cause retains why integer conversion failed.

2. C. The correct boundary is `steps <= 0`; a negative-only check accidentally accepts zero.

3. A. The minimal call removes other subsystems and directly checks the sign of the x-coordinate update.

4. D. The handler is responsible for bad commands, not for hiding faults in movement code.

5. B. A clean, complete rerun connects parser, movement, recovery, and final state to the written contract.

Find, Read, and Write Files Reliably

Reason about paths before opening them

1. B. A relative path is resolved from the process’s current working directory unless your code joins it to another explicit root.

2. C. Existence and entry kind answer different questions; a directory exists but is not a text file.

3. B. An explicit root makes the dependency visible to callers and easy to control in tests and notebooks.

Choose a reliable reading boundary

1. D. Selection prevents output or unrelated files from entering the batch; sorting makes the processing order reproducible.

2. B. Reads advance the stream cursor. Reopen or seek deliberately if the contract requires another pass.

3. A. The context manager owns cleanup; exception policy remains a separate decision.

Prevent surprising writes

1. C. Write mode truncates an existing destination on open, so compute and validate output first when possible.

2. A. Exclusive creation protects an existing path by raising `FileExistsError`.

3. D. Catch the anticipated absence specifically; unrelated failures should remain visible.

Keep Text Intact Across Encodings and Newlines

Know which side of the boundary you hold

1. A. String length counts Python’s text elements, while byte length counts the encoded integer values needed by the codec.

2. C. The exception provides evidence. Confirm the producer’s encoding rather than guessing or discarding bytes.

3. B. An explicit encoding makes the producer and consumer contract visible and portable.

Preserve meaning across representations

1. D. The utf-8-sig codec consumes a leading UTF-8 BOM while decoding ordinary UTF-8 as well.

2. B. Normalization aligns canonical Unicode representations; it is not translation, case folding, or general validation.

3. A. An empty newline setting preserves line endings for the CSV module's own record handling.

Choose evidence-preserving output behavior

1. D. A visible failure preserves evidence; any lossy continuation should be an explicit, recorded domain policy.

2. A. Creation and verification must succeed before the new artifact becomes official.

3. D. The pattern is safer than direct overwrite, but high-stakes durability needs platform-specific guarantees and testing.

Move Records Between CSV, JSON, YAML, and Python

Preserve complete CSV records

1. B. The CSV grammar distinguishes delimiters from quoted content; the module implements that distinction.

2. C. The empty newline setting lets the CSV module manage record newlines while encoding remains explicit.

3. D. CSV fields enter as text; application conversion and domain validation are separate stages.

Separate JSON syntax from application rules

1. D. The names distinguish stream/file-like input from an in-memory serialized value, not different validation levels.

2. A. The result is still a Unicode string; choose UTF-8 explicitly when writing it to bytes or a file.

3. B. Use an array or a separately specified framed format rather than silently concatenating documents.

Parse safely, then validate

1. C. Safe loading restricts construction; validation still must check the resulting value's structure and domain rules.

2. A. Grammar and application validity are separate contracts.

3. D. Choose from shape and consumers; each boundary still needs an explicit schema and validation policy.

Turn Messy Text into Trustworthy Records

Match the operation to the text question

1. A. A literal single delimiter is clearly expressed by `partition`; validate the missing or extra delimiter policy separately.

2. C. The whole field is the contract; a substring match would accept invalid surrounding characters.

3. B. Recognition and extraction precede type conversion and domain validation.

Place each rule in the right stage

1. D. Lineage makes diagnostics and audits actionable even after several transformation stages.

2. A. Idempotence makes reruns stable and prevents repeated normalization from drifting the value.

3. C. The rule depends on the relationship between two otherwise valid fields.

Keep a pipeline observable

1. A. A narrow boundary continues for known bad-input cases while preserving visibility of defects.

2. C. The evidence connects a reason to the exact input and preserves material for diagnosis or correction.

3. D. Each stage establishes assumptions needed by the next while retaining a clear failure category.

Cross the Text–Binary Boundary Safely

Know what a byte operation returns

1. B. Indexing bytes returns an integer; slicing returns another bytes value.

2. B. A codec has meaning only at a documented text boundary; arbitrary bytes are not failed text.

3. C. Binary streams enforce bytes-like input so encoding decisions remain visible.

Respect binary layout and end-of-stream

1. C. A signature is one early structural check, not complete validation or a security guarantee.

2. A. An empty bytes value means no more bytes were read.

3. B. Use an incremental decoder or preserve complete text boundaries rather than assuming chunks align with characters.

Select a bytes tool for the job

1. A. Bytearray is mutable; convert to bytes when an immutable snapshot is needed.

2. B. Subsequent reads begin at the new cursor position.

3. C. Use the simpler slice by default; optimize a demonstrated bottleneck rather than adding complexity speculatively.

Unit Challenge: Restore the Starship Signal Archive

Starship archive restoration check

1. D. A documented deterministic order stabilizes accepted records, rejections, output, and tests.

2. A. A wrapper such as xxSIG-4242yy contains a valid-looking fragment but is not a valid complete ID.

3. C. The loader is responsible for anticipated validation ValueError instances, not programming defects.

4. D. Serialization options and file encoding are separate deliberate parts of the output contract.

5. B. Verification precedes replacement so incomplete or invalid output does not become official.

Find the Right Tool Without Memorizing Everything

Read signatures and return shapes

1. A. `sorted` consumes an iterable and returns a new list; the key supplies comparison values without replacing the original items.

2. B. Parameters after the star are keyword-only, so `key=str.casefold` makes the call valid and explicit.

3. C. `removeprefix` removes one matching prefix and returns the remaining `north`.

Keep module and distribution identities separate

1. D. Installers select distributions, while Python imports modules and import packages; one distribution can expose differently named imports.

2. A. `find_spec` performs discovery without establishing origin category, quality, compatibility, or successful execution.

3. B. `importlib.metadata` reads installed distribution metadata, so its version lookup uses the distribution identity.

Make a proportionate reuse decision

1. C. The module origin tests the shadowing hypothesis directly before any broad repair changes the environment.

2. D. The nearby tools solve the requirement clearly without adding installation and maintenance decisions.

3. A. Neither random experimentation nor untested reading is enough; together they support a narrow, evidence-based use.

Turn One Script into Reusable Modules

Follow names across a module boundary

1. B. The import binds the module object as `orbit_math`; its functions and constants remain attributes of that object.

2. C. Python executes the module body to populate its namespace, so unrelated top-level effects surprise importers.

3. D. Module namespaces separate their global bindings; callers cross the boundary through imported module objects or selected names.

Choose imports and explain repeated behavior

1. A. The qualified call keeps ownership visible and avoids introducing several unannounced names.

2. B. After successful initialization, the module object is cached by import name and ordinarily reused.

3. C. A fresh process removes cached module state and hidden notebook execution order from the experiment.

Diagnose search and entry-point behavior

1. D. Those observations distinguish interpreter, location, search, shadowing, and command-mode hypotheses.

2. A. The guarded call runs for top-level execution while ordinary import can reuse the module without launching its demo or command.

3. B. `-m` takes a dotted module name, not a filename. The selected interpreter discovers it through the import system.

Design a Package People Can Rely On

Connect package files and import names

1. C. The leading dot anchors the import at the current package, so it names an owned sibling module.

2. D. Package initialization is part of importing; keep it predictable and limited to establishing the package interface.

3. A. The facade decouples the caller’s public import from the current implementation file.

Separate public promises from implementation details

1. B. The underscore is a convention about intended use and stability, not an enforced privacy mechanism.

2. C. `__all__` controls star-import selection and can signal intent; omitted attributes can still be reachable.

3. D. A deliberate small facade gives callers useful operations without turning the entire implementation into a compatibility promise.

Repair import structure without breaking callers

1. A. The package-root import is the advertised contract; updating its re-export can hide the internal move.

2. B. A lower-level owner removes the cycle rather than hiding timing symptoms.

3. C. `__main__.py` supplies the package execution route, while a callable keeps the action reusable and package import quiet.

Build a Project You Can Navigate and Share

Give each file one understandable role

1. D. `pyproject.toml` gives tools one standard project-level metadata location.

2. A. `src/` is a source-layout directory, not part of the import-package name.

3. B. The tree communicates product source separately from development verification. Later units will develop a full test layout.

Choose portable project boundaries

1. C. The separation catches accidental source-tree imports and makes installation part of the development contract.

2. D. An explicit destination makes ownership and behavior stable across notebooks, installations, and working directories.

3. A. Ignore rules normally control untracked-path reporting; tracked content requires a separate deliberate change.

Know exactly what the next commit records

1. B. The staged diff compares the index with the current commit, revealing the proposed snapshot.

2. C. Staging captures content at that moment; status and both diff views reveal later unstaged edits.

3. D. Commits form the history graph, while branch names move to identify the latest commit on a line of work.

Give Each Project a Clean Python Environment

Identify the Python that will actually run

1. A. The interpreter reports its own executable and installation prefixes; prompt text is only shell convenience.

2. B. A venv separates Python installation locations; it is not a container or general security boundary.

3. C. Declarations are portable project source; the machine-specific environment can be recreated from them.

Send commands to the intended environment

1. D. The interpreter selects and runs its pip module, avoiding ambiguity from a separate bare `pip` launcher.

2. A. Activation is shell convenience; the environment already exists and can be addressed directly.

3. B. Dependency consistency is one check; imports, project behavior, and built artifacts need separate verification.

Keep declarations and observations distinct

1. C. Declare the chosen direct relationship; the astronomy distribution records its own transitive parser requirement.

2. D. A freeze is an environment observation and may contain transitive or unrelated state; it is not a rationale.

3. A. Virtual environments are disposable and generally non-portable; declarations make reliable recreation possible.

Build and Verify an Installable Python Project

Connect metadata to its packaging role

1. B. A build frontend uses this standardized table to prepare and call the selected backend.

2. C. The build frontend supports maintainers producing artifacts; users do not need it to run this package.

3. D. Distribution and import identities serve different systems and may use different spellings.

Verify development installation without confusing it with release

1. A. Editable mode connects imports to source for development; artifact verification remains a separate step.

2. B. The installer generates a launcher that imports `constellation.cli` and invokes its `main` attribute.

3. C. Multiple launch mechanisms can share one behavior owner and remain consistent.

Inspect and verify what a user receives

1. D. A backend can produce an archive that omits code or has broken entry metadata; inspection and clean installation test the deliverable.

2. A. That combination removes working-source visibility and shows the clean interpreter imported its installed copy.

3. B. The standard frontend/backend interface lets the command request builds without embedding one backend’s implementation.

Unit Challenge: Release the Moonlight Cipher Kit

Moonlight Cipher release check

1. C. The decoder owns reusable behavior. Entry modules decide when the program action runs.

2. D. Entry-point metadata uses an importable module, a colon, and the callable attribute without parentheses.

3. A. The fresh interpreter, exact artifact, and outside working directory remove source-tree shortcuts.

4. B. The decoder is deliberate project source; environments, caches, and locally reproducible build output remain ignored.

5. C. The changed input rule belongs in the default fixture, while versioned metadata and artifact evidence must follow the changed product.

Milestone Challenge: Recover the Observatory Log

Check the reasoning behind your result

1. C. The contract requires station and reading columns. A wrong header makes the file unusable, rather than merely containing one invalid observation.

Build Your First Useful Class

Trace class calls and independent instances

1. A. Each `__init__` call assigns a new list to that instance, so changing Dawn's list does not change Dusk's list.

2. B. `is` compares identity. Equality can be defined separately, while `type` and `isinstance` ask classification questions.

3. C. A class statement creates a class object. Calling that class begins construction and returns an instance.

Follow an instance method call

1. D. Access through `tracker` binds that existing instance to the method before the explicit argument is supplied.

2. A. The class call handles construction and return. `__init__` receives and initializes the instance and returns `None`.

3. B. Instance access binds the instance. The signature needs one parameter for it plus one for the caller's prefix.

Choose state ownership deliberately

1. C. Assigning a new list to each instance during initialization prevents accidental shared mutable state.

2. D. Assignment through the ordinary instance creates an instance value; it does not rewrite the class attribute.

3. A. Cohesive evolving state plus repeated valid operations benefits from a named owner. Simple transformations and values do not require a class.

Keep Objects in Valid States

Identify valid construction

1. B. The inclusive invariant is `0 <= charge <= capacity`, so a full charge of 20 is valid.

2. C. An invariant is a lasting object rule, so every operation that can affect the fields must preserve it.

3. D. Python readers treat underscore-prefixed names as internal unless documented otherwise, but the runtime does not make them inaccessible.

Protect state during failed operations

1. A. Every condition that can reject the operation should be checked before the first mutation is committed.

2. B. A failed atomic operation leaves both participants exactly as they were.

3. C. Every public path that can affect protected state must preserve the same lasting rules.

Match behavior to an interface

1. D. A property supports value-like reading and computes from the source fields, avoiding duplicate state.

2. A. The method communicates intent and centralizes the preconditions and state transition.

3. B. A classmethod alternate constructor parses its format, then delegates construction and invariant enforcement to `cls(...)`.

Use Dataclasses for Value-Like Objects

Identify what the decorator generated

1. C. The decorator generates common field-based methods; it cannot invent the domain's valid-coordinate rules.

2. D. Annotations describe intended types but do not enforce runtime values on their own.

3. A. Default dataclass equality compares field tuples only when the two operands have the identical type.

Keep dataclass instances independent

1. B. `default_factory=list` calls `list()` for each construction that omits the field.

2. C. The two instances receive separate list objects, so mutation does not leak across them.

3. D. Dataclass field order drives the generated signature, which must obey ordinary function parameter ordering.

Decide what value-like means

1. A. Frozen instances reject normal field rebinding, but do not recursively freeze mutable objects referenced by fields.

2. B. `dataclasses.replace` constructs a changed value while leaving the frozen source untouched.

3. C. Coordinates have natural field-based value semantics. A changing unique player more naturally owns identity and transitions.

Choose Composition Before Inheritance

Trace injected collaborators

1. D. Explorer owns the workflow but delegates selection policy to the injected collaborator.

2. A. Behavior-based collaboration does not require the two implementations to share a project-defined base class.

3. B. An explicit seam allows controlled implementations to replace uncertain ones while the expedition workflow stays unchanged.

Accept or reject an inheritance relationship

1. C. The base owns shared validation and assignments; the subtype extends initialization after preserving them.

2. D. Strengthening the input precondition surprises callers that legitimately rely on all base-accepted strings.

3. A. The caller can continue using the result under the same contract even though the subtype specializes its content.

Choose the smallest honest abstraction

1. B. The tour keeps identity and stop state while an injected policy owns the varying arrangement behavior.

2. C. A pure transformation needs no owned evolving state, so a function communicates the complete contract.

3. D. A subtype must preserve caller expectations. Borrowing implementation is weaker and is often better served by a helper or collaborator.

Model a Complete Road-Crossing Game

Assign each game responsibility

1. A. Coordinates have field-based meaning and benefit from equality, representation, and non-mutating derivation.

2. C. The game owns the cross-object turn workflow and status rules while participants own their local transitions.

3. B. A class should earn a responsibility. A constrained string and lookup express this small request clearly.

Follow the deterministic turn order

1. D. Collision is checked after traffic advances, so the matching final positions produce a loss.

2. B. Player preview validates board boundaries before the game commits any part of the turn.

3. A. The immutable result separates domain behavior from presentation and makes exact assertions possible.

Change the model without tangling it

1. C. The game already coordinates collision after player and traffic transitions; the shelter qualifies that same decision.

2. D. A vehicle wraps under lane policy while a player accepts directions and board bounds, so neither is a behavioral subtype of the other.

3. A. A supplied policy isolates variation and lets scenarios use a deterministic sequence or rule.

Unit Challenge: Run the Tiny Robot Tournament

Tiny Robot Tournament design check

1. B. Energy is robot state, so Robot protects its range on every public mutation path.

2. A. Affordability is checked before mutation, so a rejected turn commits no part of the tournament transition.

3. D. Combo is a tournament workflow rule, not an intrinsic property of a move value or reusable robot.

4. B. Name, cost, and impact determine move value semantics, and accepted turns should not mutate the move description.

5. C. A subtype would put a tournament coordination rule on the wrong object and add a relationship that callers do not need.

Design a Friendly Command with argparse

From shell tokens to namespace values

1. A. The shell performs tokenization first. Quotes group the path, and Python receives the resulting argument string without the grouping quotes.

2. B. Process arguments arrive as strings. An argparse type converter may then produce an integer.

3. C. The option terminator tells the parser to treat following tokens as positional values even when they begin with hyphens.

Choose the clearest argument shape

1. D. append collects each supplied value in encounter order.

2. A. store_true implements a presence switch. type=bool would treat non-empty text such as "false" as true.

3. B. The converter validates the token’s command syntax. Application-specific rules remain in the reusable core or boundary.

Read help and failure evidence

1. C. Help is a successful process request. Argparse writes it and exits with status 0.

2. D. An unsupported choice is a command-grammar error, so argparse explains it on stderr and uses status 2 before the core runs.

3. A. The Namespace belongs to the CLI adapter. Ordinary parameters or a domain settings value keep the core usable from Python and easier to inspect.

Put Output, Errors, and Exit Status in the Right Place

Choose the correct output channel

1. B. The promised result data belongs on stdout so a pipe or redirect can consume it.

2. C. A diagnostic belongs on stderr, leaving the machine-readable stdout contract intact.

3. D. A normal pipe connects the producer’s stdout to the consumer’s stdin.

Translate failures at the boundary

1. A. One return channel should not represent both a valid count and an error signal. Let parsing raise and translate the error at the command boundary.

2. B. The entry point adapts the ordinary return value to process termination.

3. C. A missing requested file is an anticipated boundary failure the command can explain. Broad catches hide unrelated defects and process control.

Inspect direct and process evidence

1. D. A direct call checks the function boundary. A child-process check is needed for entry-point wiring and real process behavior.

2. A. An argument list avoids an extra shell boundary, improving portability and reducing injection risk.

3. B. A nonzero status may be the expected contract. check=False preserves it as data on CompletedProcess for exact assertions.

Resolve Configuration Without Exposing Secrets

Trace the winning setting

1. C. Environment is stronger than TOML, and no explicit CLI value replaces it.

2. D. False is a meaningful explicit value, not an omitted value. Testing against None preserves it.

3. A. The resolver needs to distinguish an omitted CLI option from a strongest-source value.

Convert and validate raw settings

1. B. Environment values are strings. Truthiness cannot interpret human boolean words, so accepted spellings must be converted explicitly.

2. C. A startup error makes the misspelling repairable instead of quietly falling back to another value.

3. D. Resolving relative configuration paths beside the configuration file avoids behavior changing with the launch directory.

Find the accidental disclosure

1. A. Direct command arguments may be recorded or inspected. Use an appropriate secret input or management mechanism instead.

2. B. getpass improves one interactive input channel. It does not replace storage, lifecycle, redaction, or access-control decisions.

3. C. The redacted message confirms relevant state without copying the sensitive value into logs.

Write Useful Logs Without Polluting Output

Send each message to the right mechanism

1. D. Requested result data belongs to the stdout format contract, not the configurable logging system.

2. A. The module knows the event and emits it; the application decides whether and where the DEBUG record appears.

3. B. A concise stderr diagnostic serves the user immediately without corrupting stdout. Logging may add optional operational detail.

Own configuration at the boundary

1. C. A record must pass both logger and handler filtering. INFO on the handler rejects DEBUG.

2. D. Libraries emit records through named loggers; applications own handlers, levels, destinations, and formatting.

3. A. Each attached handler can emit the same record, so repeated unowned setup commonly creates duplicate lines.

Read safe diagnostic evidence

1. B. Passing a template and arguments lets logging postpone final formatting when the level is disabled.

2. C. logger.exception includes the active exception context, so use it at a deliberate failure boundary.

3. D. The repaired record identifies the operation and whether configuration exists without copying the sensitive value.

Keep the Command Thin and the Core Reusable

Place each responsibility in the right layer

1. A. Ordinary values keep selection behavior reusable from a notebook, CLI, or another application.

2. B. Selecting and owning an external stream is adapter work. The core can consume the iterable it receives.

3. C. Only the outer entry point adapts an ordinary return value to process exit, leaving main callable directly.

Choose direct or process-level evidence

1. D. Only a process-level invocation exercises command discovery, wrapper wiring, streams, and operating-system status together.

2. A. The reusable rule has no process dependency, so a direct call provides precise evidence with little setup.

3. B. A command that works only from its source directory may rely on undeclared paths or imports. An unrelated working directory reveals that coupling.

Choose a CLI tool without outsourcing architecture

1. C. Rich focuses on terminal presentation. Keep it behind a human renderer so JSON output remains stable.

2. D. A framework may improve parsing ergonomics, but it cannot decide or enforce the application's internal responsibility boundaries automatically.

3. A. Argparse covers the command grammar with no new dependency, allowing the unit to focus on process and application boundaries.

Unit Challenge: Decode the Deep-Sea Beacon

Deep-Sea Beacon command check

1. B. Stdout is the published JSONL data channel. Operational detail belongs on the configured stderr log handler.

2. C. An explicit CLI value has stronger precedence than the environment and built-in default.

3. D. Returning preserves reuse and direct checks. `raise SystemExit(main())` belongs only at the actual process entry point.

4. A. Strict mode stops after the first rejection while preserving output already produced and returning status 1.

5. B. The subprocess crosses the real process boundary, complementing rather than replacing focused direct checks.

Write Tests That Protect Behavior

Connect assertions to the wind contract

1. A. Exact equality connects the boundary input to the one result promised by the contract. The other assertions also pass for an incorrect category.

2. B. The contract names ValueError and a stable message fragment. A broad Exception check could accept an unrelated defect.

3. C. Approximate comparison is useful for floating-point measurements with an accepted tolerance, not for exact labels, counts, or keys.

Find false confidence and brittle checks

1. D. The assertion checks the arranged input, so any returned category—including a wrong one—can pass.

2. A. The helper's existence may be an implementation choice rather than a public promise. A public input/output test survives that refactor.

3. B. Membership protects the label vocabulary but is too broad to protect the 70-kph boundary.

Choose cases and state honest conclusions

1. C. The just-below, exact, and just-above values reveal how the implementation assigns the threshold and its neighbors.

2. D. A focused test should construct the state it needs. Depending on a previous test makes isolated or reordered runs unreliable.

3. A. A passing suite supports its exercised claims in the observed environment; it does not prove untested inputs or boundaries.

Run Pytest and Read the Failure

Follow pytest discovery and selection

1. B. The full node ID uses :: between file, class, and test function and selects exactly that collected node.

2. C. Default pytest function discovery looks for test-prefixed names. The function body has not run, so production behavior is not yet relevant.

3. D. -x supports a focused investigation. A complete suite run is still needed after the repair.

Use the failure phase and assertion evidence

1. A. Collection never produced a runnable node. Package installation and import context come before behavior assertions.

2. B. The test body has not run. Setup needs a fixture whose name matches the requested argument.

3. C. A direct comparison preserves the expected record shape and gives assertion rewriting useful subexpression evidence.

Choose the smallest useful rerun

1. D. Collection-only output answers whether nodes exist without mixing that question with their behavior.

2. A. The focused node directly tests the repair hypothesis. Follow it with the file and complete suite.

3. B. A local change can affect other contracts. The broader rerun checks those already-declared expectations.

Turn Examples into Clear Parametrized Cases

Turn contract rows into readable nodes

1. C. Just below, exactly at, and just above 70 reveal which interval owns the boundary and its immediate neighbors.

2. D. The descriptive ID states why the row exists and what the contract promises, so a failed node is understandable by itself.

3. A. Each parameter row becomes a separately reported and selectable test case.

Keep case data and assertions clear

1. B. Shared decision logic can make both sides agree on the same wrong threshold. Explicit contract values are independent and reviewable.

2. C. Separate nodes expose the failed row, allow focused reruns, and let later cases run independently.

3. D. Separating distinct contracts avoids branches inside the test and makes each failed node’s intent obvious.

Select interactions without creating noise

1. A. The 45-kph input triggers amber while 0.9-km visibility triggers red, so the row specifically tests priority.

2. B. Parameter values are passed as supplied, not automatically copied. Shared mutable data can leak state between cases.

3. C. A meaningful regression name records why a formerly failing input must remain protected.

Build an Isolated Test World with Fixtures

Read fixture requests and fresh state

1. D. Pytest resolves a fixture from the requested argument, calls it within the lifecycle, and passes its result to the test.

2. A. Function scope creates a fixture instance for each requesting test, preventing that list mutation from leaking.

3. B. Behavior-defining case data should remain close to the assertion so a reader can understand the failure.

Choose safe fixture lifetimes

1. C. The fixture resumes after the test call to release or restore the resource even when the call failed.

2. D. Immutable shared reference data avoids mutation leaks. Mutable process or file state should usually have narrower isolation.

3. A. Start with the nearest understandable owner. Move it to a shared conftest.py only when multiple nearby test modules need it.

Isolate files, output, and logs

1. B. The test exercises real local file behavior while keeping paths isolated from the repository and other tests.

2. C. capsys returns separate text captured from Python’s stdout and stderr streams and resets its interval.

3. D. Structured record assertions protect the diagnostic contract without coupling to formatter details that are not promised.

Control Dependencies Without Fragile Mocks

Control time, randomness, and sending

1. A. An explicit clock seam makes the result deterministic while ordinary callers can still use the real UTC clock.

2. B. The in-memory collaborator performs useful sender behavior and records it without external effects.

3. C. The explicit chooser seam controls exactly the operation the function needs and avoids order-dependent global random state.

Patch state and collaborator lookups safely

1. D. The fixture records whether the key was absent or had another value and restores that state automatically.

2. A. Patch where publish looks up the name: the binding inside the alerts module.

3. B. A spec catches some imaginary-method mistakes, though the test still must choose meaningful behavioral assertions.

Protect obligations without freezing internals

1. C. The returned value is public behavior and remains valid if internal steps are safely reorganized.

2. D. An interaction assertion should represent a promised effect, not an incidental implementation sequence.

3. A. A temporary real filesystem boundary is fast, faithful, and simpler than reproducing Path behavior with mocks.

Test at the Smallest Boundary That Can Fail

Match risks to the smallest revealing boundary

1. B. Entry-point metadata and process lookup exist only at the installed process boundary; direct calls bypass them.

2. C. A real isolated file exercises encoding and filesystem behavior without writing into the repository.

3. D. Names are hints; the boundary comes from which real parts collaborate during the test.

Combine focused, integration, and process evidence

1. A. The direct adapter test connects more application parts and streams while remaining inside the current process.

2. B. A smoke route gives broad but shallow evidence for one journey; detailed cases still need focused tests.

3. C. The layers answer different questions: focused tests give cheap, diagnostic depth while process tests cover selected wiring risks.

Turn coverage and flaky failures into questions

1. D. Statement coverage records execution, not assertion quality, missing requirements, boundary correctness, or all decision outcomes.

2. A. Branch evidence distinguishes visiting a decision line from exercising each possible destination.

3. B. Controlling the time dependency removes the race and makes the expected date deterministic.

Preserve Bugs and Explore General Rules

Reduce failures and begin with useful red evidence

1. C. A small counterexample keeps the same cause while making the violated normalization rule obvious and fast to reproduce.

2. D. TDD red should fail for the promised behavior, not unrelated setup. Establish a valid test environment before changing the feature.

3. A. The regression remains executable evidence that the discovered boundary must not return.

Express general rules with generated inputs

1. B. Idempotence means applying the operation again does not change the already-normalized result.

2. C. A strategy that directly describes valid inputs explores the contract efficiently and makes the domain reviewable.

3. D. Examples and properties provide different evidence: examples explain named behavior while generated tests explore a wider input space.

Use a minimal counterexample without overclaiming

1. A. Shrinking removes irrelevant complexity while preserving the failure, making diagnosis and communication easier.

2. B. Applying the same buggy normalization twice can be stable. Add an invariant about allowed whitespace to represent the real promise.

3. C. The property continues broad exploration while the named regression communicates the specific historical obligation.

Unit Challenge: Clear the Buggy Spaceport for Launch

Clear the Buggy Spaceport for Launch readiness check

1. D. Python treats booleans as integers for isinstance checks, so the public mass contract needs an explicit Boolean rejection.

2. A. A controlled clock produces one exact deterministic value; no timing tolerance or race is needed.

3. B. The public contract promises every record round-trips; reading the real temporary file exposes the slice that dropped one.

4. C. The command’s channel contract reserves stdout for successful fee data and stderr for invalid-input diagnostics.

5. D. Coverage is execution evidence and a prompt for questions; it is not a proof of correctness or completeness.

Write Python People Can Follow

Names that carry meaning

1. A. The name exposes both the represented duration and the unit used by the calculation.

2. B. Uppercase words separated by underscores communicate a project constant; the unit also remains visible.

3. C. The condition reads as a domain statement and avoids a double negative or redundant comparison.

Layout that reveals the program

1. D. A formatter can apply layout rules, but it cannot know which real-world value and unit `t` represents.

2. A. Each rejected case finishes, so the reader does not carry several open conditions into the main path.

3. B. Separate responsibilities are useful when they change for different reasons and can be checked independently.

A change another maintainer can verify

1. C. Statements and intermediate names help when a reader must verify multiple independent rules.

2. D. The printed-card constraint explains why the value exists and what must remain synchronized.

3. A. The changed threshold needs examples on both sides so the intended behavior is explicit before the implementation moves.

Make Contracts Visible with Type Annotations

What annotations do—and do not do

1. B. Python stores the annotation, but the runtime still receives the supplied object unless the program performs validation.

2. C. `-> None` makes the action-only contract explicit and lets a checker reject callers that treat the result as a value.

3. D. The empty expression supplies no item whose type can be inferred, so the annotation records the intended contents.

Honest result and collection shapes

1. A. The union makes absence an explicit case that callers must handle.

2. B. `Iterable[int]` promises exactly the operation used: producing integers during one traversal.

3. C. Some iterables are single-use; materialize once or require a stronger reusable collection contract.

Structured records at a real boundary

1. D. TypedDict describes dictionary structure to type checkers; it does not replace the runtime dictionary or validate external data.

2. A. Runtime validation establishes the facts that make the narrower static contract honest.

3. B. A dataclass models a runtime object; TypedDict is best when dictionary shape itself is the intended representation.

Read MyPy Errors and Close Type Gaps

Follow MyPy’s evidence

1. C. The numeric contract is coherent; trace the string back to the external boundary and make the conversion explicit.

2. D. The code names the category and supports targeted documentation or a narrowly scoped ignore.

3. A. It reports the checker’s inferred type, helping trace how a later diagnostic was reached.

Close uncertain paths deliberately

1. B. The branch removes the None case, leaving the string member of the declared union.

2. C. `object` accepts any value but retains static restrictions; `Any` permits unchecked operations to spread.

3. D. Operations on Any generally produce more Any, which can hide mistakes far from the original boundary.

Keep the checker trustworthy

1. A. The implementation conflicts with a coherent text-label contract; repairing the value preserves the useful interface.

2. B. A code-specific ignore documents the exact gap and can be reported when it becomes unnecessary.

3. C. A clean static run is one bounded kind of evidence, not proof of runtime behavior or complete correctness.

Format and Lint a Project with Ruff

Read the Ruff report

1. D. F401 is a bounded static finding. Review whether the import is unnecessary or an intentional public re-export.

2. A. The selected path and rule narrow the lint run while leaving the file unchanged.

3. B. Domain responsibility and public error policy require knowledge beyond mechanical source rules.

Turn findings into a reviewed patch

1. C. `ruff format --check` reports and exits nonzero for files that would be reformatted while leaving them unchanged.

2. D. Run rewrite-producing lint fixes first, then let the formatter normalize their final layout.

3. A. Unsafe fixes require human judgment and behavior evidence; availability alone is not permission.

Make Ruff reproducible

1. B. Target version constrains version-sensitive rules and formatting to syntax the project promises to support.

2. C. A deliberate list keeps policy reviewable when Ruff evolves.

3. D. Different versions, config sources, paths, or overrides can legitimately produce different checks.

Catch Small Problems Before Commit

Read a hook configuration

1. A. The command already owns its target; appended filenames could change scope or produce a confusing invocation.

2. B. System hooks reuse installed commands, so the project environment must provide compatible versions.

3. C. The repository shares configuration, while each checkout installs its own Git hook script.

Respond to hook evidence

1. D. The hook produced a new working-tree state; review and stage it, then prove a clean idempotent rerun.

2. A. Skipped is a selection result, not evidence about files outside the hook’s current target.

3. B. An ordinary commit checks staged changes; all-files establishes the baseline across existing tracked content.

Know what the local gate proves

1. C. CI adds a committed workflow and clean shared environment instead of relying on one checkout.

2. D. The gate should report formatting drift with a nonzero exit status, not rewrite uncommitted files.

3. A. A version update changes executable tooling and must be reviewed and verified like another dependency change.

Repeat the Quality Gate in CI

Trace the workflow from event to result

1. B. The job executes on a hosted runner matching that label; Python is selected in a later setup step.

2. C. Steps share one job workspace and must establish the inputs required by later commands.

3. D. The final true command exits zero, hiding the failure signal automation needs.

Find the earliest useful CI evidence

1. A. The install step cannot see project metadata; establish repository location/state before editing application behavior.

2. B. CI adds an environmental comparison; enumerate observed differences instead of assuming randomness.

3. C. Checkout, setup, installation, and checks form a dependency chain; repair the first broken prerequisite.

Design a defensible shared gate

1. D. The evidence is bounded by the exact workflow, commit/ref, jobs, tools, and environments that ran.

2. A. A matrix should answer a support-compatibility question, not add combinations without purpose.

3. B. One committed policy prevents editor, hook, local command, and CI from drifting into separate rule sets.

Refactor in Small, Verified Steps

Separate structure from behavior

1. C. Callers can observe a different score, so the change needs new expected behavior rather than being hidden inside a refactor.

2. D. The test freezes current observable behavior so refactoring and a future behavior decision do not become one ambiguous patch.

3. A. A before/after ownership assertion observes the boundary that output-only checks miss.

Choose one structural improvement

1. B. Scoring and presentation have different reasons to change and can be checked through separate interfaces.

2. C. A named constant is valuable when its domain meaning or shared policy reduces ambiguity.

3. D. Exception order can be part of observed behavior even when every individual validation rule remains.

Review the refactor as evidence

1. A. A focused patch lets reviewers compare the same behavior against one understandable structural move.

2. B. Side-effect order is observable; choose and test the new behavior explicitly rather than hiding it in a refactor.

3. C. Maintainability balances local clarity with navigation and abstraction cost.

Unit Challenge: Untangle the Clockwork Observatory

Clockwork Observatory readiness check

1. D. Negating the numeric score produces descending score under normal ascending sort; the name remains ascending.

2. A. The contract caps the one-point-per-minute penalty at 60.

3. B. The before/after equality assertion directly observes whether ranking changed caller-owned records.

4. C. CI should report formatting drift with a nonzero status, not rewrite files on the runner.

5. D. The complete evidence chain covers the stated behavior plus the new static and reproducibility contracts.

Design Documentation Around Reader Needs

Match the question to the document

1. A. A tutorial controls a safe learning path and produces an early successful result.

2. B. A how-to assumes orientation and guides the reader toward a particular practical result.

3. C. Reference describes the public interface precisely for a reader working with it now.

Decide what the quick-start evidence proves

1. D. Compilation checks syntax; it does not import the package or compare behavior.

2. A. A fresh environment exposes missing installation steps and hidden local state while the result checks the public promise.

3. B. Expected output lets a reader distinguish success from a command that merely exits without an obvious error.

Choose what belongs at the project entrance

1. C. The descriptive text names the task while the target remains encoded in the Markdown link destination.

2. D. Reference can carry exact API detail while the README remains a short entry path with a descriptive link.

3. A. A clean rerun reveals and then verifies the prerequisite that a new reader needs.

Write Instructions People Can Follow

Prepare the reader before step 1

1. B. The reader can locate the directory by a project file and then reproduce the relative path.

2. C. The command and comparison give the reader an observable readiness check.

3. D. The opening promises a safe learning result and states what prior knowledge is not needed.

Make each step verifiable

1. A. The named old/new value and unchanged inputs preserve the connection between edit and result.

2. B. The complete domain value lets the reader verify the exhibit identity and changed score.

3. C. The error already shows that relative path resolution began from the wrong directory.

Make the page scannable and accessible

1. D. The text communicates the relationship the diagram contributes to this lesson.

2. A. The link names both the action and destination subject.

3. B. The heading names the exact failure and the purpose of the section.

Document a Public Python API

Put each fact where readers look for it

1. C. `int` does not communicate a unit; the public contract should.

2. D. Explanation can develop rationale and trade-offs without crowding reference.

3. A. The underscore and lack of public use signal an implementation detail rather than a caller promise.

Document facts types cannot carry

1. B. Integer division counts two complete periods and ignores the remaining 15 minutes.

2. C. A type narrows representation but does not supply unit or valid domain.

3. D. Replacement and encoding affect caller data and safe use.

Read the interface a caller receives

1. A. Built-in help uses Python’s documentation system for the object exposed at runtime.

2. B. Import-time prompts or effects run when an introspection-based tool imports the module.

3. C. The public behavior and public documentation must be reconciled through an explicit contract decision.

Keep Small Examples Honest with Doctest

Confirm that executable examples actually ran

1. D. A successful silent run and a run that found nothing can both lack useful collection evidence.

2. A. Attempted count verifies collection; failed count verifies comparison.

3. B. The collection option tells pytest to discover doctest examples in modules.

Control textual variation without hiding behavior

1. C. Sorting the displayed view preserves exact membership while avoiding an order promise the set does not make.

2. D. An all-ellipsis string can accept almost any non-empty content and protects no meaningful field.

3. A. The option can express that tokens matter while incidental whitespace layout does not.

Choose the clearest executable evidence

1. B. The short public transcript is both readable documentation and useful executable evidence.

2. C. Parametrization states the boundary table clearly without crowding a public example.

3. D. No attempted examples means there was nothing for comparison to validate.

Review Documentation Like Code

Make the gate prove collection

1. A. The universal condition has no counterexample in an empty iterable, so collection needs its own assertion.

2. B. The attempted count proves discovery; the failed count proves comparison.

3. C. A path relative to the committed test file is reproducible across machines and shells.

Keep reader examples and checks connected

1. D. The subprocess exercises the same public artifact and process boundary the reader uses.

2. A. The test names the same relative-path base promised by the instructions.

3. B. Independent copies create multiple sources whose behavior and output can diverge.

Combine automated and human evidence

1. A. Existence cannot judge relevance, findability, or clarity.

2. D. A generated file can exist while its content or route remains unusable.

3. C. Public usage remains stable, so the README should not depend on the private implementation name.

Build a Documentation Site with Quarto

Repair the source, not generated output

1. B. QMD is the maintained source; the generated HTML is replaceable output.

2. C. The declared sidebar content order controls visible navigation.

3. D. `_site` is generated from maintained source, configuration, and assets.

Make site navigation answer reader questions

1. A. A new learner needs a safe first success before exact lookup material.

2. B. The source-relative QMD link is portable and the label states the destination’s purpose.

3. C. Navigation still needs to orient readers who lack product vocabulary.

Use the build mode that fits the decision

1. D. Global configuration can affect every page, and deployment needs a complete artifact.

2. A. Preview shortens local feedback but does not replace the deployment build.

3. B. A successful process can still render an incomplete target set; named outputs protect the intended artifact.

Publish Documentation and Keep It Trustworthy

Connect the reviewed source to the public artifact

1. C. The generated site directory is the artifact produced and checked by the render.

2. D. The build provides review evidence while publication remains tied to the accepted branch.

3. A. Equivalent, visible contracts let maintainers reproduce failures before pushing.

Locate the earliest failing boundary

1. B. The renderer is absent, so no documentation source has been processed yet.

2. C. The failure sits between source selection/render and the expected artifact set.

3. D. Secrets must use the platform’s protected secret/permission mechanisms, never committed text.

Update documentation with public change

1. A. The public name appears across caller-facing contracts and executable examples.

2. B. Published routes can be compatibility promises and may need deliberate migration.

3. C. Build recency and content accuracy are different facts.

Unit Challenge: Publish the Puzzle Garden Field Guide

Puzzle Garden publication check

1. D. Both discovery and comparison are visible in the attempted/failed pair.

2. A. The reader already knows the package and needs one task-oriented guide.

3. B. Offset one changes `L M B` to `M B L` before names are decoded.

4. C. Resolution is automatable; task relevance still needs reader review.

5. D. The verified `_site` directory is the generated public artifact.

Milestone Challenge: Publish a Tiny Community Library Tool

Check the reasoning behind your result

1. D. The contract covers both failure reporting and preservation of stored data. Test both with an isolated temporary catalog.

Study record

Record a date when you can explain the unit's ideas and have worked through its challenge. This is a personal study record, not a verified grade or certificate.

Unit	Lessons reviewed	Challenge attempted	Revisit
0. Get Started			
1. Learning to Read and Write Python Code			
2. Numbers, Text, and Other Values			
3. Organizing and Traversing Collections			
4. Making Decisions and Repeating Work			
5. Building Reusable Functions			
6. Sharing, Changing, and Copying Objects			
7. Solving Problems with Algorithms			
8. Errors, Exceptions, and Debugging			
9. Files, Paths, and External Data			
10. Modules, Environments, and Python Projects			
11. Object-Oriented Python and Dataclasses			
12. Command-Line Applications			
13. Testing Python Programs			
14. Code Quality and Maintainability			
15. Documentation as Part of the Product			

License and source

Course sources: github.com/freecampus/python. Report corrections there or through the website.

BSD 3-Clause License

Copyright (c) 2026, Free Campus

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Font licenses

This book embeds DejaVu fonts and renamed glyph subsets from the Noto family. The Noto subsets retain their original copyright notices and are distributed under the SIL Open Font License below.

Format: https://www.debian.org/doc/packaging-manuals/copyright-format/1.0/ Upstream-Name: DejaVu fonts Upstream-Author: Stepan Roh src@users.sourceforge.net (original author), see /usr/share/doc/fonts-dejavu-core/AUTHORS for full list Source: https://dejavu-fonts.github.io/

Files: * Copyright: Copyright (c) 2003 by Bitstream, Inc. All Rights Reserved. Bitstream Vera is a trademark of Bitstream, Inc. DejaVu changes are in public domain. License: bitstream-vera Permission is hereby granted, free of charge, to any person obtaining a copy of the fonts accompanying this license ("Fonts") and associated documentation files (the "Font Software"), to reproduce and distribute the Font Software, including without limitation the rights to use, copy, merge, publish, distribute, and/or sell copies of the Font Software, and to permit persons to whom the Font Software is furnished to do so, subject to the following conditions: . The above copyright and trademark notices and this permission notice shall be included in all copies of one or more of the Font Software typefaces. . The Font Software may be modified, altered, or added to, and in particular the designs of glyphs or characters in the Fonts may be modified and additional glyphs or characters may be added to the Fonts, only if the fonts are renamed to names not containing either the words "Bitstream" or the word "Vera". . This License becomes null and void to the extent applicable to Fonts or Font Software that has been modified and is distributed under the "Bitstream Vera" names. . The Font Software may be sold as part of a larger software package but no copy of one or more of the Font Software typefaces may be sold by itself. . THE FONT SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT OF COPYRIGHT, PATENT, TRADEMARK, OR OTHER RIGHT. IN NO EVENT SHALL BITSTREAM OR THE GNOME FOUNDATION BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, INCLUDING ANY GENERAL, SPECIAL, INDIRECT, INCIDENTAL, OR CONSEQUENTIAL DAMAGES, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF THE USE OR INABILITY TO USE THE FONT SOFTWARE OR FROM OTHER DEALINGS IN THE FONT SOFTWARE. . Except as contained in this notice, the names of Gnome, the Gnome Foundation, and Bitstream Inc., shall not be used in advertising or otherwise to promote the sale, use or other dealings in this Font Software without prior written authorization from the Gnome Foundation or Bitstream Inc., respectively. For further information, contact: fonts at gnome dot org.

© 2014-2021 Adobe (http://www.adobe.com/), with Reserved Font Name "Source".

This Font Software is licensed under the SIL Open Font License, Version 1.1.

This license is copied below, and is also available with a FAQ at: http://scripts.sil.org/OFL

SIL OPEN FONT LICENSE Version 1.1 - 26 February 2007

PREAMBLE The goals of the Open Font License (OFL) are to stimulate worldwide development of collaborative font projects, to support the font creation efforts of academic and linguistic communities, and to provide a free and open framework in which fonts may be shared and improved in partnership with others.

The OFL allows the licensed fonts to be used, studied, modified and redistributed freely as long as they are not sold by themselves. The fonts, including any derivative works, can be bundled, embedded, redistributed and/or sold with any software provided that any reserved names are not used by derivative works. The fonts and derivatives, however, cannot be released under any other type of

license. The requirement for fonts to remain under this license does not apply to any document created using the fonts or their derivatives.

DEFINITIONS "Font Software" refers to the set of files released by the Copyright Holder(s) under this license and clearly marked as such. This may include source files, build scripts and documentation.

"Reserved Font Name" refers to any names specified as such after the copyright statement(s).

"Original Version" refers to the collection of Font Software components as distributed by the Copyright Holder(s).

"Modified Version" refers to any derivative made by adding to, deleting, or substituting – in part or in whole – any of the components of the Original Version, by changing formats or by porting the Font Software to a new environment.

"Author" refers to any designer, engineer, programmer, technical writer or other person who contributed to the Font Software.

PERMISSION & CONDITIONS Permission is hereby granted, free of charge, to any person obtaining a copy of the Font Software, to use, study, copy, merge, embed, modify, redistribute, and sell modified and unmodified copies of the Font Software, subject to the following conditions:

1 Neither the Font Software nor any of its individual components, in Original or Modified Versions, may be sold by itself.

2 Original or Modified Versions of the Font Software may be bundled, redistributed and/or sold with any software, provided that each copy contains the above copyright notice and this license. These can be included either as stand-alone text files, human-readable headers or in the appropriate machine-readable metadata fields within text or binary files as long as those fields can be easily viewed by the user.

3 No Modified Version of the Font Software may use the Reserved Font Name(s) unless explicit written permission is granted by the corresponding Copyright Holder. This restriction only applies to the primary font name as presented to the users.

4 The name(s) of the Copyright Holder(s) or the Author(s) of the Font Software shall not be used to promote, endorse or advertise any Modified Version, except to acknowledge the contribution(s) of the Copyright Holder(s) and the Author(s) or with their explicit written permission.

5 The Font Software, modified or unmodified, in part or in whole, must be distributed entirely under this license, and must not be distributed under any other license. The requirement for fonts to remain under this license does not apply to any document created using the Font Software.

TERMINATION This license becomes null and void if any of the above conditions are not met.

DISCLAIMER THE FONT SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT OF COPYRIGHT, PATENT, TRADEMARK, OR OTHER RIGHT. IN NO EVENT SHALL THE COPYRIGHT HOLDER BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, INCLUDING ANY GENERAL, SPECIAL, INDIRECT, INCIDENTAL, OR CONSEQUENTIAL DAMAGES, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF THE USE OR INABILITY TO USE THE FONT SOFTWARE OR FROM OTHER DEALINGS IN THE FONT SOFTWARE.

Copyright 2013 Google LLC

This Font Software is licensed under the SIL Open Font License, Version 1.1. This license is copied below, and is also available with a FAQ at: https://scripts.sil.org/OFL

SIL OPEN FONT LICENSE Version 1.1 - 26 February 2007

PREAMBLE The goals of the Open Font License (OFL) are to stimulate worldwide development of collaborative font projects, to support the font creation efforts of academic and linguistic communities, and to provide a free and open framework in which fonts may be shared and improved in partnership with others.

The OFL allows the licensed fonts to be used, studied, modified and redistributed freely as long as they are not sold by themselves. The fonts, including any derivative works, can be bundled, embedded, redistributed and/or sold with any software provided that any reserved names are not used by derivative works. The fonts and derivatives, however, cannot be released under any other type of license. The requirement for fonts to remain under this license does not apply to any document created using the fonts or their derivatives.

DEFINITIONS "Font Software" refers to the set of files released by the Copyright Holder(s) under this license and clearly marked as such. This may include source files, build scripts and documentation.

"Reserved Font Name" refers to any names specified as such after the copyright statement(s).

"Original Version" refers to the collection of Font Software components as distributed by the Copyright Holder(s).

"Modified Version" refers to any derivative made by adding to, deleting, or substituting – in part or in whole – any of the components of the Original Version, by changing formats or by porting the Font Software to a new environment.

"Author" refers to any designer, engineer, programmer, technical writer or other person who contributed to the Font Software.

PERMISSION & CONDITIONS Permission is hereby granted, free of charge, to any person obtaining a copy of the Font Software, to use, study, copy, merge, embed, modify, redistribute, and sell modified and unmodified copies of the Font Software, subject to the following conditions:

1 Neither the Font Software nor any of its individual components, in Original or Modified Versions, may be sold by itself.

2 Original or Modified Versions of the Font Software may be bundled, redistributed and/or sold with any software, provided that each copy contains the above copyright notice and this license. These can be included either as stand-alone text files, human-readable headers or in the appropriate machine-readable metadata fields within text or binary files as long as those fields can be easily viewed by the user.

3 No Modified Version of the Font Software may use the Reserved Font Name(s) unless explicit written permission is granted by the corresponding Copyright Holder. This restriction only applies to the primary font name as presented to the users.

4 The name(s) of the Copyright Holder(s) or the Author(s) of the Font Software shall not be used to promote, endorse or advertise any Modified Version, except to acknowledge the contribution(s) of the Copyright Holder(s) and the Author(s) or with their explicit written permission.

5 The Font Software, modified or unmodified, in part or in whole, must be distributed entirely under this license, and must not be distributed under any other license. The requirement for fonts to remain under this license does not apply to any document created using the Font Software.

TERMINATION This license becomes null and void if any of the above conditions are not met.

DISCLAIMER THE FONT SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT OF COPYRIGHT, PATENT, TRADEMARK, OR OTHER RIGHT. IN NO EVENT SHALL THE COPYRIGHT HOLDER BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, INCLUDING ANY GENERAL, SPECIAL, INDIRECT, INCIDENTAL, OR CONSEQUENTIAL DAMAGES, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF THE USE OR INABILITY TO USE THE FONT SOFTWARE OR FROM OTHER DEALINGS IN THE FONT SOFTWARE.