

Day 25 Polish and reflect

Self-Study Guide · About 60 minutes

By the end you will be able to

- Fix failing tests and the bugs you find by using your program, adding a test for each bug.
- Clean up the project with a formatter, a linter, and a type checker.
- Write a README that says what the project does, how to install it, and how to run it.
- Look back at your Day 1 code and your notes, and choose three things to learn next.

What to remember from earlier

- To find a bug, reproduce it as a failing test, narrow down where the values go wrong, fix the cause, then run every test. ([Day 20: Testing and debugging, 20.6 Finding a Bug Systematically](#))
- `ruff format` . fixes the layout of every file, and `ruff check` . reports likely bugs and style problems. ([Day 19: Type hints and code style, 19.7 Formatting and Linting with ruff or black](#))
- A type checker such as `mypy` reads your code without running it and reports every value that does not match its hint. ([Day 19: Type hints and code style, 19.4 Checking Types with mypy or pyright](#))
- `requirements.txt` lists a project's packages, and `python -m pip install -r requirements.txt` installs them all. ([Day 13: Modules, packages and environments, 13.10 Installing Packages with pip](#))
- `python hello.py`, typed in a terminal in the file's folder, runs a script. ([Day 1: Setup and your first program, 1.3 Writing and Running a Script](#))

The last day. Open your project folder, activate its virtual environment, and reread yesterday's recap in `notes.txt`, especially the list of anything that failed. The feature freeze from Day 24 still holds: today adds **no new features**. You make what exists solid, make it readable to someone else, and then look back at how far you have come since Day 1.

Today's hour, roughly:

Milestone	Minutes
25.1 Fix tests and bugs	15
25.2 Formatter, linter, checker	10
25.3 Write a README	15
25.4 Look back at Day 1	10
25.5 Choose what to learn next	10

25.1 Fixing Failing Tests and Bugs

Start with the tests. Run `python -m pytest`. For each failure, read the report from the bottom up (Day 20) and answer one question before changing anything: **is the code wrong, or is the test**

wrong? Usually it is the code. Sometimes the test expects something the brief never asked for, or has a mistake of its own, such as a wrong total worked out by hand. Fix whichever one disagrees with your Day 23 description, never both at once, and run all the tests again.

Then **use your program as a stranger would**, for ten minutes. Day 12 called this testing every path: try every menu choice or option, in an unusual order, with the empty file, with bad input, and with the largest input you have. Whenever something goes wrong, follow Day 20's steps:

1. **Reproduce it**, and write it down as a failing test if the bug is in a logic function.
2. **Narrow it down**: `print` or `breakpoint()` until you find the first place where a value differs from what you expected.
3. **Fix the cause**, then run every test. Keep the new test.

A test written for a bug you fixed is called a **regression test**: it fails if the bug ever comes back (a **regression** is something that used to work and has broken again). It is the most valuable test in your file, because it guards against a mistake you have already proved you can make.

WATCH OUT

Fixing a test by making it agree with the bug. When `test_total_by_month` expects 42.5 and the code returns 30.0, it is tempting to change the 42.5. Work out the right answer by hand first. If 42.5 is right, the test has just found a bug, and changing it throws that discovery away.

PAUSE AND TRY

Question 1 Multiple choice

Ana's quiz game test `assert score_percent(7, 10) == 70` fails: pytest reports that `score_percent(7, 10)` returned `0.7`. Her Day 23 description says the game shows "your score as a percentage". What should she do?

- A. Change the test to expect `0.7`, so the test passes
- B. Delete the test, because the function works
- C. Fix `score_percent` to return 70, then run every test
- D. Change the test to `assert score_percent(7, 10) > 0`

25.2 Formatter, Linter, and Type Checker

Next, let the tools from Day 19 read the whole project. Install them into the project's environment if they are not there yet, then run each one from the project folder:

```
python -m pip install ruff mpy
ruff format .
ruff check .
mypy .
```

(black and pyright, from Day 19, work just as well.) The dot means "this folder and every folder inside it"; both tools skip `.venv` for you.

- `ruff format .` rewrites the layout. Nothing to decide; just look at what changed.

- `ruff check .` lists likely bugs and style problems, each with a rule code. `ruff check --fix .` fixes the safe ones. Read every remaining one and decide: an unused import goes; an unused variable is often a half-finished idea or a misspelled name.
- `mypy .` reports every value that does not match its hint. Day 23's stubs promised a return type for every function; any stub you never filled in shows up here as missing a return. Fix the code when the hint describes what the function should do, and fix the hint when it was a wrong guess.

When all three are quiet, **run `python -m pytest` again**. A formatter never changes what code does, but fixes for the linter and the type checker do, and the tests are how you know nothing broke.

KEY IDEA

Format, lint, type-check, then run the tests again. The tools find problems by reading; the tests prove the fixes did not change what the program does.

PAUSE AND TRY

Question 2 Multiple choice

Ben runs `ruff check .`, which reports

F841 Local variable 'total' is assigned to but never used inside his `summary()` function.

What is the best first step?

- A. Delete the line that assigns `total`, without reading the function
- B. Read `summary()` to find out whether `total` was meant to be used, for example in the return value
- C. Run `ruff format .`, which fixes rule F841
- D. Ignore it, because the program runs without an error

25.3 Writing a README

A project that only you can run is not finished. A **README** is a short file at the top of a project folder that tells a newcomer what the project is and how to use it. Its usual name is `README.md`: the `.md` means it is written in **Markdown**, a plain-text layout where a line starting with `#` is a heading, `##` a smaller heading, and a line starting with `-` a bullet. Code-hosting sites such as GitHub and most editors (including VS Code's preview) show a README formatted, but it reads well as plain text too.

A good README answers three questions, in this order:

1. **What does it do?** One or two sentences, adapted from your Day 23 description, and one example of it running.
2. **How do I install it?** Every command from a fresh download to a working environment, on each system you know about.
3. **How do I run it?** The command to start it, the options or menu choices, and how to run the tests.

Here is the expense tracker's. Commands that the reader types sit on their own lines, indented by four spaces, which Markdown shows as code:

Expense tracker

A command-line program that records what you spend and shows the total for each category and each month. Expenses are saved in `expenses.csv`, so they are kept between runs.

Install

You need Python 3.12 or newer. In the project folder:

```
python -m venv .venv
source .venv/bin/activate
python -m pip install -r requirements.txt
```

On Windows, activate with `.venv\Scripts\Activate.ps1` instead.

Run

```
python main.py
```

Choose from the menu: 1 adds an expense, 2 lists them all, 3 shows totals by category, 4 shows totals by month, 5 quits.

Test

```
python -m pytest
```

Then **check it the hard way**: follow your own README word for word, as if you had never seen the project. Better still, delete `.venv`, rebuild it from the Install section, and run the program and the tests. Every command you had to add or fix along the way is one a reader would have been stuck on. Check, too, that `requirements.txt` lists everything the project imports from outside the standard library (Day 13).

PAUSE AND TRY

Question 3 Short answer

Sam's weather tool README says only: "Install: run `python weather.py CITY`." Name two things a newcomer would be missing, and say what Sam should add.

25.4 Looking Back at Day 1

Open your `day01` folder and read `hello.py`, your first script, and the other files you wrote on Day 1. Then read your recap notes from the start of the course to now, all of them, not just the last few.

Day 1's first program was two `print` lines, and its biggest danger was a missing quote. Since then you have learned to store and change values, to make decisions and repeat work, to organize data in lists, dictionaries, and classes, to split a program into functions and modules, to save data in files, to recover from errors, to fetch data from the internet, and to prove your code works with tests. Your capstone uses nearly all of it, in a project you planned yourself.

Write a short reflection at the end of `notes.txt`, three or four sentences:

- Something from your early notes that confused you and is now easy. What made it click?
- The part of the capstone you are proudest of, and the part you would rebuild differently.
- Which day you would reread first if you had to teach this course to a friend.

Looking back like this is not a nice extra. Noticing **how** you learned (which days stuck, which ideas needed a second try, whether typing every example helped) is what lets you learn the next thing faster, without a course to set the pace.

25.5 What to Learn Next

Finish by writing down **three things to learn next**, at the bottom of `notes.txt`. Make each one concrete: not "get better at Python" but "build a small website for my expense tracker with Flask" or "solve the first five Advent of Code puzzles". Choose at least one that grows directly out of your capstone.

Here are the main directions people go after a first course, with a good starting point for each:

- **Web development:** Flask or FastAPI for small web apps and APIs, then Django for large sites with accounts and databases.
- **Data analysis:** pandas for tables of data, Matplotlib for charts, and Jupyter notebooks, which mix code, results, and notes on one page.
- **Automation:** scripts that rename and sort files, collect data from websites (web scraping, always within a site's terms of use), and run on a schedule. The standard library's `pathlib`, `shutil`, and `csv` go a long way.
- **Deeper Python:** writing your own context managers (the machinery behind `with`), `async` and `await` for programs that wait on many things at once, packaging a project so others can install it with pip, and Python's data model, the full set of special methods that Day 17's `__eq__` and `__lt__` belong to.
- **Practice:** Exercism's Python track for short exercises with feedback, or Advent of Code puzzles, a new set released each December and fun to solve in any month.

Whatever you choose, keep the habits from this course: an hour at a time, type the code, predict before you run, test what you build, and write down what confused you.

KEY IDEA

Pick three concrete next steps and write them down. The habits matter more than the topic: small daily sessions, typing and predicting, testing, and keeping notes.

Go further

The Python Tutorial, which each day's Go further section pointed to, ends with a chapter on where to go after it: The Python Tutorial: What Now?. For Markdown, GitHub's Basic writing and formatting syntax covers everything a README needs.

Self-check

Check each statement. If the answer is "not yet," reread the linked milestone and give it ten more minutes. When every box is checked, you have finished Learn Python in 25 Days: from `print("Hello, world!")` to a planned, tested, documented program of your own. Well done.

- ☐ Every test passes, and I added a regression test for each bug I fixed in a logic function. Reread: 25.1 Fixing Failing Tests and Bugs
- ☐ I used my program as a stranger would and fixed what went wrong, without adding features. Reread: 25.1 Fixing Failing Tests and Bugs
- ☐ The formatter, linter, and type checker report nothing, and the tests still pass afterward. Reread: 25.2 Formatter, Linter, and Type Checker
- ☐ My README says what the project does, how to install it, and how to run it and its tests. Reread: 25.3 Writing a README
- ☐ I followed my README from a fresh virtual environment and it worked. Reread: 25.3 Writing a README
- ☐ I reread my Day 1 code and my notes, and wrote a short reflection. Reread: 25.4 Looking Back at Day 1
- ☐ I wrote down three concrete things to learn next. Reread: 25.5 What to Learn Next