

# Day 24 Build

Self-Study Guide · About 60 minutes · With answer key

## By the end you will be able to

- Build the remaining must have features one at a time, running the program after each one.
- Handle bad input and missing or damaged files with a clear message instead of a crash.
- Write at least five pytest tests for the project's core logic.
- Stop adding features at the end of the hour and write down what is left.

## What to remember from earlier

- `raise ValueError(message)` stops a function and hands the problem to whoever called it. ([Day 15: Errors and exceptions, 15.7 Raising Your Own Errors with raise](#))
- `except SomeError as error` catches only that type of exception and stores it in `error`. ([Day 15: Errors and exceptions, 15.4 Catching Specific Exceptions](#))
- Opening a file inside `try` and catching `FileNotFoundError` covers the real moment of opening, which a check with `exists()` cannot. ([Day 15: Errors and exceptions, 15.9 Ask Forgiveness, Not Permission](#))
- `pytest` finds files whose names start with `test_` and runs each function in them whose name starts with `test`. ([Day 20: Testing and debugging, 20.3 Running Tests with pytest](#))
- Edge cases such as an empty list, zero, or a value on a boundary are where bugs usually hide. ([Day 20: Testing and debugging, 20.4 Normal Cases and Edge Cases](#))
- `with pytest.raises(ValueError):` passes only if the code inside it raises that exception. ([Day 20: Testing and debugging, 20.5 Testing Expected Exceptions](#))
- A failed or slow web request raises an exception, so wrap it in `try` and always pass a `timeout`. ([Day 22: Working with web data, 22.7 Timeouts and Network Errors](#))

Open your project folder from Day 23, activate its virtual environment (look for `(.venv)` in the prompt), and reread the feature list and recap in `notes.txt`. Today is the main building day: by the end of the hour, every must have feature works, the program survives bad input, and at least five tests check its logic.

As on Day 23, there is no new reading and no reference solution. Each milestone names the day that taught what it needs.

Today's hour, roughly:

| Milestone                              | Minutes |
|----------------------------------------|---------|
| 24.1 The remaining must haves          | 30      |
| 24.2 Bad input                         | 8       |
| 24.3 Missing files                     | 7       |
| 24.4 At least five tests               | 12      |
| 24.5 Stop, and write down what is left | 3       |

## 24.1 One Feature at a Time

Work through your must have list **in order**, one feature at a time. For each one:

1. **Fill in** the stubs this feature needs, and connect them to `main()` (a menu choice or a command-line option).
2. **Run the program** and use the new feature the way a user would, with ordinary input.
3. **Try one more case**: the empty list, the first expense of a new month, the last question in the bank.
4. **Tick it off** in `notes.txt`, then start the next feature.

Run after **every** feature, not after all of them. When the program worked five minutes ago and does not now, the bug is almost certainly in the lines you just wrote, which is a far smaller place to search than the whole program. That is Day 20's systematic search, made easy by never letting much change between runs. If a bug still hides, use the steps from Day 20: reproduce it, narrow it down, and look inside with `breakpoint()` or your editor's debugger.

Reach for the standard library before writing a utility yourself (Day 21). Totals by category are a `defaultdict(float)` or a `Counter`; grouping by month is a key such as `expense.day.strftime("%Y-%m")`; the most common words or pages are `Counter.most_common`; command-line options are `argparse`; dates are `datetime`.

#### KEY IDEA

Build one feature, run it, try it, tick it off, then start the next. Running after every feature keeps each new bug inside the few lines you just wrote.

#### WATCH OUT

**Building three features, then running.** It feels faster to write all the remaining features and test once at the end. Then the first run shows a traceback from one feature, the fix uncovers a second bug in another, and the hour goes on untangling them. One feature per run is slower for five minutes and faster for the hour.

## 24.2 Handling Bad Input

So far you have typed only good input. Real users, including you next week, will not. List every place where data **enters** your program, and decide what each one does with something it cannot use:

| Where data enters              | What can go wrong                         | Tool (day)                                                               |
|--------------------------------|-------------------------------------------|--------------------------------------------------------------------------|
| <code>input()</code>           | A word where a number belongs, a bad date | <code>try / except ValueError</code> , ask again (Day 15)                |
| Command-line options           | A missing or wrongly typed option         | <code>argparse</code> with <code>type=int</code> checks for you (Day 21) |
| A value your logic must refuse | A negative amount, an empty title         | <code>raise ValueError</code> with a message (Day 15)                    |

|                            |                                          |                                                           |
|----------------------------|------------------------------------------|-----------------------------------------------------------|
| The network (weather tool) | No connection, a slow server, a bad city | timeout=,<br>except<br>requests.RequestException (Day 22) |
|----------------------------|------------------------------------------|-----------------------------------------------------------|

A good pattern splits the job in two, as Day 15 did. The **logic** function refuses bad values by raising an exception with a message that says what was wrong. The code that **talks to the user** catches it, prints the message, and carries on (asks again, or goes back to the menu). The logic function never prints, so your tests in 24.4 can check that it refuses, and the program never crashes, because every exception that bad input can cause is caught where the input came in.

### WORKED EXAMPLE Refusing a bad amount

The expense tracker's `parse_amount` stub from Day 23, filled in. The loop below it plays the part of `main()`, trying four typed amounts:

`parse_amount.py`

```

1 def parse_amount(text: str) -> float:
2     """Return the amount typed as text, in dollars.
3
4     Raises:
5         ValueError: If text is not a number, or is 0 or less.
6     """
7     amount = float(text)
8     if amount <= 0:
9         raise ValueError(f"the amount must be more than 0, not {amount}")
10    return amount
11
12
13 for typed in ["12.50", "abc", "-3", "7"]:
14     try:
15         print(parse_amount(typed))
16     except ValueError as error:
17         print("Not saved:", error)

```

```

12.5
Not saved: could not convert string to float: 'abc'
Not saved: the amount must be more than 0, not -3.0
7.0

```

- `float("abc")` raises `ValueError` by itself; the function does not need to check for words.
- `float("-3")` works, but a negative expense makes no sense, so line 9 raises a `ValueError` of its own, with a message in the same style.
- One `except ValueError` in the caller handles both kinds of problem, and `error` carries the message to print. (Older Python versions may word `float()`'s message differently.)

**Test it:** for every prompt and option, try a word, an empty line, a negative number, and zero. None of them may crash the program or change your saved data.

**PAUSE AND TRY****Question 1** Tracing

In `parse_amount.py`, the list on line 13 is changed to `["0", "2.5.1", " 4 "]`. What does the program print? (`float()` ignores spaces at both ends of its text.)

## 24.3 Handling Missing and Damaged Files

Every project reads a file or the network, and two things go wrong with files that have nothing to do with the user's typing:

- **The file is missing.** On the very first run there is no save file yet. That is normal, and the program should simply start empty. For a file the user names (the log analyzer's input), a missing file is a mistake to report clearly, for example with `sys.exit(f"No file called {name}.")` (Day 13).
- **The file is damaged.** A save file can be half-written if the computer lost power, or edited by hand and broken. JSON that cannot be read raises `json.JSONDecodeError`; a CSV row with a word in the amount column makes `float()` raise `ValueError` when you rebuild the object.

Day 15's advice applies: open the file inside `try` and catch the specific exceptions, rather than checking with `exists()` first.

**WORKED EXAMPLE** Loading high scores safely

The quiz game loads its high scores at the start. One function covers all three cases: a good file, no file, and a damaged one.

`load_scores.py`

```
1 import json
2 from pathlib import Path
3
4
5 def load_high_scores(path: Path) -> dict[str, int]:
6     """Return the saved high scores, or {} if there are none to load."""
7     try:
8         with open(path, encoding="utf-8") as file:
9             return json.load(file)
10    except FileNotFoundError:
11        return {}
12    except json.JSONDecodeError:
13        print(f"Warning: {path} is damaged, so the high scores start again.")
14        return {}
15
16
17 print(load_high_scores(Path("good_scores.txt")))
18 print(load_high_scores(Path("no_such_file.json")))
19 print(load_high_scores(Path("broken_scores.txt")))
```

`good_scores.txt` holds `{"Ana": 9, "Ben": 7}`, `no_such_file.json` does not exist, and `broken_scores.txt` stops halfway: `{"Ana": 9, "Ben": .` The output:

```
{'Ana': 9, 'Ben': 7}
{}
Warning: broken_scores.txt is damaged, so the high scores start again.
{}
```

- A missing file is normal, so it returns an empty dictionary without a word.
- A damaged file is not normal, so the user gets a warning before the scores start again. Silently losing data is worse than losing it with an explanation.
- Each `except` names one exception, so a real bug elsewhere in the function (a misspelled name, say) still shows its traceback instead of hiding behind a friendly message.

**Test it:** run the program with the save file deleted, then with a save file you broke on purpose in your editor (delete its last few characters). Both runs must start without a traceback. Keep a copy of your real data before you break anything.

### PAUSE AND TRY

#### Question 2 Multiple choice

A classmate's version catches only `FileNotFoundError`:

`load_scores_crash.py`

```
1 import json
2 from pathlib import Path
3
4
5 def load_high_scores(path: Path) -> dict[str, int]:
6     """Return the saved high scores, or {} if the file does not exist."""
7     try:
8         with open(path, encoding="utf-8") as file:
9             return json.load(file)
10    except FileNotFoundError:
11        return {}
12
13
14 print("Loading...")
15 print(load_high_scores(Path("broken_scores.txt")))
```

What happens when it runs, with the same damaged `broken_scores.txt`?

- It prints `Loading...` and then `{}`
- It prints `Loading...`, then stops with a traceback ending in `JSONDecodeError`
- It prints `Loading...` and then `{'Ana': 9}`, the part of the file that could be read
- It stops with `FileNotFoundError` before printing anything

## 24.4 Testing the Core Logic

Now prove the logic works, with at least **five** pytest tests (Day 20). Install pytest into the project's environment and add it to `requirements.txt` so anyone rebuilding the project gets it too (Day 13):

```
python -m pip install pytest
python -m pip freeze > requirements.txt
```

Test the **logic functions**: the ones that take data in and return data out. They need no typing and no network, so their tests run in a fraction of a second and give the same answer every time. Leave `main()` and anything that calls `input()` to your own hands-on testing. For the weather tool, test the functions that turn a saved response into lines of text, as Day 22's report module allowed. Choose the five or more from Day 20's kinds of case:

| Kind of case (Day 20) | Expense tracker example                                         |
|-----------------------|-----------------------------------------------------------------|
| Normal case           | Two food expenses and one transport expense give two totals     |
| Empty                 | No expenses give an empty dictionary of totals                  |
| Boundary              | An amount of exactly 0 is refused; 0.01 is accepted             |
| Bad input             | <code>parse_amount("abc")</code> raises <code>ValueError</code> |
| A bug you found today | Whatever went wrong in 24.1, so that it can never come back     |

The test file goes beside the modules, named so `pytest` finds it. Here is the start of the expense tracker's (no output is shown, because yours will test your own functions):

```
1 from datetime import date
2
3 import pytest
4
5 from expenses import Expense, parse_amount, total_by_category
6
7
8 def test_parse_amount_reads_a_number() -> None:
9     assert parse_amount("12.50") == 12.5
10
11
12 def test_parse_amount_refuses_zero() -> None:
13     with pytest.raises(ValueError):
14         parse_amount("0")
15
16
17 def test_total_by_category_with_no_expenses() -> None:
18     assert total_by_category([]) == {}
19
20
21 def test_total_by_category_adds_each_category() -> None:
22     expenses = [
23         Expense(date(2026, 10, 1), 12.5, "food"),
24         Expense(date(2026, 10, 2), 4.25, "transport"),
25         Expense(date(2026, 10, 3), 3.0, "food"),
26     ]
27     assert total_by_category(expenses) == {"food": 15.5, "transport": 4.25}
```

Run them with `python -m pytest` (or `python -m pytest -v` for one line per test). A test that fails is doing its job: read the failure from the bottom up, as Day 20 showed, and decide whether the code or the test is wrong. The amounts here are chosen so the sums are exact; Day 2 showed that

floats such as 0.1 are close, not exact, so `0.1 + 0.2 == 0.3` would fail for a reason that has nothing to do with your code.

#### WATCH OUT

**Testing a function that prints or asks.** A test cannot type, so a function that calls `input()` fails under `pytest` instead of waiting for an answer, and a function that only prints returns `None`, so there is nothing to `assert` about. If the function you most want to test talks to the user, split it: move the calculation into a logic function that returns a value, test that, and let `main()` print the result.

#### PAUSE AND TRY

##### Question 3 Short answer

Mia's log analyzer has `count_errors(lines: list[str]) -> int`, which returns how many lines contain the word `ERROR`. List five tests worth writing for it, each with its input and the expected result, and say which kind of case each one is.

## 24.5 Stopping on Time

At the end of the hour, **stop adding features**, even if one is half done and you can see exactly how to finish it. Software teams call this a **feature freeze**: a point after which no new features are added, only fixes, so that what exists can be made solid. Day 25 is for fixing, cleaning up, and writing about the project, and it needs the whole hour.

Before you close the editor:

1. If a feature is half built, either finish it in two minutes or disconnect it from `main()`, so the program still runs.
2. Run the program once more and run `python -m pytest`. Write down anything that fails; you fix it tomorrow.
3. Move every unfinished feature and every new idea to the **nice to have** list in `notes.txt`. It is not lost, only waiting.
4. Write today's recap: what works, what broke and how you found it, and what you would do differently.

#### KEY IDEA

A feature freeze stops new features so the existing ones can be made solid. A smaller program that works is the goal; unfinished ideas go on the nice to have list.

## Go further

Day 24 has no new reading. If a milestone sent you back, these are good second explanations: The Python Tutorial: Errors and Exceptions and the `pytest` documentation's Get Started page.

## Self-check

Check each statement. If the answer is "not yet," reread the linked milestone and the day it names. Anything still failing goes in your notes for Day 25.

- ☐ I built my must have features one at a time and ran the program after each one. Reread: 24.1 One Feature at a Time
- ☐ Every place where data enters my program handles bad input with a message instead of a crash. Reread: 24.2 Handling Bad Input
- ☐ My logic functions raise exceptions for bad values, and the code that talks to the user catches them. Reread: 24.2 Handling Bad Input
- ☐ My program starts cleanly when its save file is missing, and warns me when the file is damaged. Reread: 24.3 Handling Missing and Damaged Files
- ☐ I wrote at least five pytest tests for my logic functions, including an edge case and an expected exception. Reread: 24.4 Testing the Core Logic
- ☐ I stopped adding features at the end of the hour and moved unfinished ideas to my nice to have list. Reread: 24.5 Stopping on Time

## Check Your Reasoning

Answers and explanations for every question, numbered to match.

### Question 1

```
Not saved: the amount must be more than 0, not 0.0
Not saved: could not convert string to float: '2.5.1'
4.0
```

- `float("0")` is `0.0`, and `0.0 <= 0` is `True`, so line 9 raises the function's own `ValueError`. Zero is a boundary (Day 20): the rule says "more than 0", so 0 itself is refused.
- `"2.5.1"` is not a number, so `float()` raises `ValueError` with its own message before line 8 runs.
- `" 4 "` becomes `4.0`: `float()` ignores the spaces, and `4.0` is more than 0.

Reread 24.2 *Handling Bad Input* if your output differed.

### Question 2

**B.** The file exists, so `open` works and `FileNotFoundError` is never raised. `json.load` then raises `json.decoder.JSONDecodeError`, which no `except` clause names, so it travels up (Day 15) and stops the program with a traceback, after `Loading...` has already been printed.

- **A** is wrong: `{}` is returned only from the `except FileNotFoundError` clause, which does not match this exception.
- **C** is wrong: `json.load` never returns part of a file. It either reads valid JSON completely or raises an exception.
- **D** is wrong: the file is there, and line 14 prints `Loading...` before the file is even opened.

Reread 24.3 *Handling Missing and Damaged Files* if you chose differently.

### Question 3

Model answer (other good sets exist):

| Input                                                    | Expected | Kind of case                         |
|----------------------------------------------------------|----------|--------------------------------------|
| <code>["ERROR disk full", "INFO ok", "ERROR net"]</code> | 2        | Normal                               |
| <code>[]</code>                                          | 0        | Empty                                |
| <code>["INFO ok", "INFO started"]</code>                 | 0        | No matches                           |
| <code>["ERROR one"]</code>                               | 1        | One item                             |
| <code>["INFO ok", "WARN x", "ERROR last"]</code>         | 1        | Position: the match is the last line |

A sixth worth adding is a line such as `"error in lower case"`, which forces Mia to decide whether case matters and then write that decision down as a test.

A correct answer gives five inputs with exact expected numbers and covers at least a normal case, the empty list, and a list with no matches.

Reread 24.4 *Testing the Core Logic* if your answer missed one of those.