

Day 20 Testing and debugging

Self-Study Guide · About 85 minutes

By the end you will be able to

- Write automated tests with `assert`, and run them with `pytest` using file and function names it can find.
- Choose test cases for normal values, edge cases, and expected exceptions, and test an exception with `pytest.raises`.
- Find a bug systematically with `print`, `breakpoint()`, and an editor's debugger, stepping through code and inspecting variables.
- Use logging to report what a program is doing, at a level you can turn up or down.

What to remember from earlier

- Test a function by calling it with inputs whose answers you already know, especially the values on each side of a boundary. ([Day 9: Functions, part 1, 9.6 Testing a Function](#))
- Read a traceback from the bottom up. The last line says what went wrong, and the frames above it say where. ([Day 15: Errors and exceptions, 15.1 Reading a Traceback from the Bottom Up](#))
- `raise ValueError("message")` stops a function and sends an exception to its caller. ([Day 15: Errors and exceptions, 15.7 Raising Your Own Errors with raise](#))
- Code under `if __name__ == "__main__":` runs when you run the file, but not when another file imports it. ([Day 13: Modules, packages and environments, 13.5 Code That Runs Only When Run Directly](#))
- With a virtual environment active, `python -m pip install name` installs a package into it. ([Day 13: Modules, packages and environments, 13.10 Installing Packages with pip](#))
- A `with` statement runs its block and then cleans up, such as closing a file, even if the block raised an exception. ([Day 14: Files and data formats, 14.2 The with Statement](#))
- A docstring's `Raises` section says which exceptions a function raises and when. ([Day 19: Type hints and code style, 19.8 Docstrings and Comments That Help](#))

Make a `day20` folder for today's files, open it in your editor, and run each script from its terminal, as on earlier days. In 20.3 you will install `pytest`, a testing tool, into a virtual environment, so create one in `day20` now if you like (Day 13: `python -m venv .venv`, then activate it).

Yesterday's type checker could tell you that a function returns an `int`. It could not tell you whether it returns the **right** `int`. Today you learn to prove that: you will write small programs that check your other programs, every time you change them, in a second. And when a check fails, you will learn to find the cause on purpose, by looking at what the program is really doing, instead of by guessing.

20.1 Why Automated Tests

A museum charges 5 dollars for children under 12, 7 for visitors aged 65 or over, and 10 for everyone else. On Day 9 you would have checked a function like this by printing each result next to the answer you expect:

print_checks.py

```
1 def ticket_price(age: int) -> int:
2     """Return the ticket price in dollars for a visitor of this age."""
3     if age < 12:
4         return 5
5     if age > 65:
6         return 7
7     return 10
8
9
10 print(ticket_price(8), "expected 5")
11 print(ticket_price(11), "expected 5")
12 print(ticket_price(12), "expected 10")
13 print(ticket_price(40), "expected 10")
14 print(ticket_price(65), "expected 7")
15 print(ticket_price(80), "expected 7")
```

```
5 expected 5
5 expected 5
10 expected 10
10 expected 10
10 expected 7
7 expected 7
```

One line is wrong: a 65-year-old pays 10, not 7, because line 5 says `age > 65` where the rule says "65 or over". Did you spot it straight away? With six lines you probably did. Now imagine sixty, or a program with twenty functions, checked again after every change. On Day 12 you tested the to-do manager by typing a sixteen-row session by hand. Each time you change the program, a change can break something that used to work, and you would have to type the whole session again, and read every line, to know.

That is the job of an **automated test**: code that runs your code with inputs whose correct answers you already know, compares what comes back with those answers, and reports only the ones that do not match. Each input with its expected answer is a **test case**. A computer never gets bored, never skims, and runs a hundred test cases in less time than it takes you to read one. Because tests cost nothing to rerun, you can run them after every change. A bug that breaks something that used to work is called a **regression**, and catching regressions the moment they happen is the biggest reason programmers write tests.

Not every check makes a good test case. A good one has these qualities:

Quality	What it means	Why it matters
A known answer	The expected result comes from the rule, worked out by hand, not from running the code	Copying the code's own output into a test would "prove" that a bug is right
One behavior	It checks one thing, such as the senior price	When it fails, its name tells you what broke

Repeatable	Same result on every run: no <code>input()</code> , no <code>random</code> , no clock	A test that sometimes fails cannot tell you whether the code changed
Independent and fast	It sets up its own data and does not rely on another test running first	You can run any test alone, and all of them often

KEY IDEA

An automated test runs your code on test cases whose answers you worked out from the rule, and reports any that do not match. Tests cost nothing to rerun, so they catch regressions the moment a change breaks something.

PAUSE AND TRY**Question 1** Multiple choice

`shipping_cost(total)` should return 0 for orders of 50 or more, and 4.99 for smaller orders. Which is a good test case?

- A. Call `shipping_cost(50)` and print the result
- B. Check that `shipping_cost(50)` returns 0
- C. Check that `shipping_cost(50)` returns the same value as `shipping_cost(50)`
- D. Ask the user to type a total with `input()`, and check the result

20.2 Checking with assert

The print checks needed a person to compare two numbers. Python can do the comparing itself, with the **assert statement**:

```
1 assert condition
2 assert condition, message
```

- `assert` is a keyword. The **condition** is any expression that is true or false, usually a comparison such as `ticket_price(65) == 7`.
- If the condition is true, nothing happens and the program carries on. A passing check is silent.
- If it is false, Python raises an **AssertionError**, the exception for "a check failed". Like any uncaught exception (Day 15), it stops the program with a traceback.
- The optional **message**, after a comma, becomes the exception's message. Use it to say what went wrong.

```
>>> assert 2 + 2 == 4
>>> assert 2 + 2 == 5
Traceback (most recent call last):
...
AssertionError
>>> price = 10
>>> assert price == 7, f"expected 7, got {price}"
Traceback (most recent call last):
...
AssertionError: expected 7, got 10
```

The first `assert` passes, so the REPL shows nothing. The second fails, and the traceback ends in a bare `AssertionError`. The third adds a message, built with an f-string, which says exactly what was expected and what came back.

WORKED EXAMPLE Turning the print checks into asserts

Here is the same buggy function, with each print check rewritten as an `assert`:

ticket_asserts.py

```
1 def ticket_price(age: int) -> int:
2     """Return the ticket price in dollars for a visitor of this age."""
3     if age < 12:
4         return 5
5     if age > 65:
6         return 7
7     return 10
8
9
10 assert ticket_price(8) == 5
11 assert ticket_price(11) == 5
12 assert ticket_price(12) == 10
13 assert ticket_price(40) == 10
14 assert ticket_price(65) == 7
15 assert ticket_price(80) == 7
16 print("All ticket checks passed")
```

```
Traceback (most recent call last):
  File ".../day20/ticket_asserts.py", line 14, in <module>
    assert ticket_price(65) == 7
           ^^^^^^^^^^^^^^^^^^^^^
AssertionError
```

- Lines 10 to 13 pass silently: 8 and 11 are under 12, and 12 and 40 fall through to `return 10`.
- Line 14 calls `ticket_price(65)`. 65 is not under 12, and `65 > 65` is false, so it returns 10. `10 == 7` is false, so `assert` raises `AssertionError`, and the traceback points straight at line 14 (read it from the bottom up, Day 15).
- Line 15 and the `print` never run, because the exception stopped the program.

Nobody had to read six lines and compare numbers. Fix line 5 to `if age >= 65:` and run it again:

ticket_asserts_fixed.py

```
1 def ticket_price(age: int) -> int:
2     """Return the ticket price in dollars for a visitor of this age."""
3     if age < 12:
4         return 5
5     if age >= 65:
6         return 7
7     return 10
8
9
10 assert ticket_price(8) == 5
11 assert ticket_price(11) == 5
12 assert ticket_price(12) == 10
13 assert ticket_price(40) == 10
14 assert ticket_price(65) == 7
15 assert ticket_price(80) == 7
16 print("All ticket checks passed")
```

All ticket checks passed

`assert` is for checking **your own code**: in tests, and for facts that must be true if the program is correct. It is not for checking what a user typed or what a file contains. Python has a switch, `python -O program.py` ("optimize"), that skips every `assert` statement, so a check written with `assert` can silently disappear. For bad input, raise an exception yourself, as `BankAccount` does with `raise ValueError(...)` (Day 15).

KEY IDEA

`assert condition, message` does nothing when the condition is true, and raises `AssertionError` with the message when it is false. Use it to check your own code, never to validate input.

WATCH OUT

Wrapping the condition and message in parentheses. This looks like a function call, but `assert` is a statement:

assert_tuple.py

```
1 def ticket_price(age: int) -> int:
2     """Return the ticket price in dollars for a visitor of this age."""
3     if age < 12:
4         return 5
5     if age > 65:
6         return 7
7     return 10
8
9
10 assert (ticket_price(65) == 7, "seniors pay 7")
11 print("All ticket checks passed")
```

All ticket checks passed

The function still has the `age > 65` bug, yet the check passes. The parentheses make one value, a **tuple** of two items (Day 7), and a non-empty tuple is always truthy (Day 4), so this `assert` can never fail. Python 3.14 prints a `SyntaxWarning` saying the assertion is always true; older versions may say nothing. Write it without parentheses:

```
assert ticket_price(65) == 7, "seniors pay 7".
```

PAUSE AND TRY

Question 2 Tracing

What does this program print? The docstring states the rule, so read it carefully.

`assert_trace.py`

```
1 def discount(price: float) -> float:
2     """Return the price after 10% off orders of 100 or more."""
3     if price > 100:
4         return price * 0.9
5     return price
6
7
8 print("Checking")
9 assert discount(50) == 50
10 assert discount(200) == 180
11 print("Halfway")
12 assert discount(100) == 90, f"discount(100) gave {discount(100)}"
13 print("Done")
```

Question 3 Short answer

Rewrite these print checks as `assert` statements, each with a message that shows the actual result. `is_teen` is the function from Day 9.

```
1 print(is_teen(12), "expected False")
2 print(is_teen(13), "expected True")
3 print(is_teen(19), "expected True")
```

20.3 Running Tests with pytest

A file full of `assert` lines has two problems. It stops at the **first** failure, so one broken check hides every check after it. And the checks sit next to the code they test, so every run of the program also runs the tests. Programmers fix both by putting the tests in a separate file, each one in its own function, and letting a tool run them.

That tool is usually **pytest**, the most widely used testing tool for Python. It is a third-party package, so install it into your project's virtual environment with pip (Day 13):

```
python -m pip install pytest
```

pytest finds your tests by their names. This is called **test discovery**:

- It looks in the current folder, and the folders inside it, for files named `test_*.py` (or `*_test.py`), such as `test_tickets.py`.
- In each such file, it runs every function whose name starts with `test`, such as `test_senior_price`. A function like that is a **test function**: it takes no arguments, sets up what it needs, and checks the result with plain `assert`.

WORKED EXAMPLE A module and its test file

Put the fixed function in a module of its own, `tickets.py`:

`tickets/tickets.py`

```
1 """Ticket prices for the town museum."""
2
3
4 def ticket_price(age: int) -> int:
5     """Return the ticket price in dollars for a visitor of this age.
6
7     Children under 12 pay 5, visitors aged 65 or over pay 7, and
8     everyone else pays 10.
9     """
10    if age < 12:
11        return 5
12    if age >= 65:
13        return 7
14    return 10
```

The tests go beside it in `test_tickets.py`, which imports the function like any module (Day 13):

tickets/test_tickets.py

```
1 from tickets import ticket_price
2
3
4 def test_child_price() -> None:
5     assert ticket_price(8) == 5
6
7
8 def test_adult_price() -> None:
9     assert ticket_price(40) == 10
10
11
12 def test_senior_price() -> None:
13     assert ticket_price(65) == 7
14
15
16 def test_child_boundary() -> None:
17     assert ticket_price(11) == 5
18     assert ticket_price(12) == 10
19
20
21 if __name__ == "__main__":
22     test_child_price()
23     test_adult_price()
24     test_senior_price()
25     test_child_boundary()
26     print("All tests passed")
```

- Each test function checks one behavior, and its name says which. Test functions return nothing, so their hint is `-> None` (Day 19).
- `test_child_boundary` checks both sides of the line at 12 (Day 9's boundary rule). Two asserts in one test are fine when they check one behavior.
- pytest does not need lines 21 to 26: it finds and calls the test functions itself, and when it imports the file, `__name__` is not `"__main__"` (Day 13). They are here so you can also run the file with plain Python, before pytest is installed:

```
All tests passed
```

To run the tests, open a terminal in the folder holding both files and type:

```
python -m pytest
```

(Typing just `pytest` also works while the environment is active.) Here is what pytest prints. This is **sample output**: the exact layout, timings, and the header lines naming your system and versions, which are left out here, vary.

```

===== test session starts =====
collected 4 items

test_tickets.py .... [100%]

===== 4 passed in 0.01s =====

```

collected 4 items says it found four test functions. Each dot is one test that passed, and the last line sums up. Add `-v` ("verbose"), as in `python -m pytest -v`, to see one line per test, with its name and PASSED. `python -m pytest test_tickets.py` runs just one file.

Now put the bug back (`age > 65` in `tickets.py`) and run `pytest` again (sample output):

```

===== test session starts =====
collected 4 items

test_tickets.py ..F. [100%]

===== FAILURES =====
_____ test_senior_price _____

    def test_senior_price() -> None:
>     assert ticket_price(65) == 7
E     assert 10 == 7
E     + where 10 = ticket_price(65)

test_tickets.py:13: AssertionError
===== short test summary info =====
FAILED test_tickets.py::test_senior_price - assert 10 == 7
===== 1 failed, 3 passed in 0.03s =====

```

Read it from the bottom up, like a traceback:

Part	Meaning
1 failed, 3 passed	The summary: one test failed, and the other three still ran and passed
FAILED test_tickets.py::test_senior_price	Which test failed: the file, two colons, then the test function
test_tickets.py:13: AssertionError	The file and line of the failing assert
E assert 10 == 7	The comparison with real values: the left side was 10 , but should be 7
E + where 10 = ticket_price(65)	Where the 10 came from
>	Marks the failing line in the test's code
..F.	One mark per test, in order: . passed, F failed

Notice the line `E assert 10 == 7`. A plain `assert` without a message says only `AssertionError`, but `pytest` rewrites your asserts so a failure shows the values on each side. That is why test functions under `pytest` seldom need messages. And the other three tests all ran: one failure no longer hides the rest.

KEY IDEA

Put tests in files named `test_*.py`, as functions named `test_*` that check with plain `assert`. `python -m pytest` finds and runs them all, prints a dot for each pass, and for each failure shows the line and the actual values.

WATCH OUT

Naming a test so pytest cannot find it. A function called `check_senior_price`, or a file called `tickets_tests.py`, is never collected. `pytest` does not warn you: it reports `collected 3 items` instead of 4, or, if no file matches, `no tests ran`. A test that never runs can never fail, so a bug in the code it was meant to check goes unnoticed. Start every test file name and test function name with `test`, and glance at the `collected` count.

PAUSE AND TRY**Question 4** Multiple choice

Which file and function will `pytest` find and run?

- A. File `cart_tests.py`, function `test_total`
- B. File `test_cart.py`, function `total_test`
- C. File `test_cart.py`, function `test_total`
- D. File `cart.py`, function `test_total`

Question 5 Short answer

Here is part of a `pytest` report (sample output). Which test failed, on which line, what did the function return, and what was expected? The rule is that orders of 50 or more ship free. Is the bug more likely in the test or in `shipping_cost`?

```
def test_free_at_fifty() -> None:
>     assert shipping_cost(50) == 0
E     assert 4.99 == 0
E         + where 4.99 = shipping_cost(50)

test_shipping.py:9: AssertionError
```

20.4 Normal Cases and Edge Cases

Which inputs should your tests use? You cannot test every possible input, so choose the ones most likely to reveal a mistake. Day 9 gave the first rule: test each side of every boundary. Here is the fuller list:

Kind of case	What it is	Examples for a list of scores
Normal case	An ordinary, everyday input	[72, 88, 95]
Boundary	The values on each side of a line the rule draws	A score of 59 and 60 if 60 is a pass
Empty, zero, and one	The smallest inputs: an empty list or string, 0, a single item	[], [0], [75]
Negative and unusual values	Values that are allowed but easy to forget	[-5, -2], a tie, a repeated value
Position	The important item first, last, or in the middle	The highest score at the start or end
Bad input	Input the function should refuse, by raising an exception	A score of 105 (20.5)

The cases other than normal ones are called **edge cases**: inputs at the edges of what a function handles, where it is most likely to break. Code written with one typical input in mind usually works for that input. Bugs live at the edges.

WORKED EXAMPLE An edge case that finds a bug

`largest` should return the largest number in a list. Its author tested it with ordinary lists, and all passed. The last test is an edge case:

`largest_tests.py`

```

1 def largest(numbers: list[int]) -> int:
2     """Return the largest number in a list that is not empty."""
3     best = 0
4     for number in numbers:
5         if number > best:
6             best = number
7     return best
8
9
10 def test_typical() -> None:
11     assert largest([3, 9, 4]) == 9
12
13
14 def test_largest_at_either_end() -> None:
15     assert largest([9, 3, 4]) == 9
16     assert largest([3, 4, 9]) == 9
17
18
19 def test_one_item() -> None:
20     assert largest([7]) == 7
21
22
23 def test_all_negative() -> None:
24     assert largest([-5, -2, -9]) == -2

```

```

25
26
27 test_typical()
28 test_largest_at_either_end()
29 test_one_item()
30 test_all_negative()
31 print("All tests passed")

```

```

Traceback (most recent call last):
  File ".../day20/largest_tests.py", line 30, in <module>
    test_all_negative()
    ~~~~~^
  File ".../day20/largest_tests.py", line 24, in test_all_negative
    assert largest([-5, -2, -9]) == -2
           ^^^^^^^^^^^^^^^^^^^^^^^^^
AssertionError

```

(The calls at the bottom stand in for `pytest` so this file runs with plain Python.) The first three tests pass. Trace the fourth: `best` starts at 0 (line 3). Is -5 greater than 0? No. -2? No. -9? No. So `best` is never replaced, and `largest` returns 0, a number that is not even in the list. Every normal test used positive numbers, which are all greater than 0, so they could never notice that 0 was a bad starting point. Start from an item that really is in the list: `best = numbers[0]`, as on Day 5. Then all four tests pass. (Python's built-in `max()` does this job already; writing `largest` yourself shows how an edge case exposes a hidden assumption.)

Each test should check **one behavior**, and its name should say which, such as `test_all_negative`. Then the report's list of failures reads like a list of what is broken. Most tests also follow the same three steps, often called **arrange, act, assert**: set up the data, call the code being tested, and check the result.

KEY IDEA

Test a normal case, then the edge cases: each side of every boundary; empty, zero, and one item; negative and unusual values; the important item in each position; and bad input. Bugs hide at the edges, where a typical input never goes.

WATCH OUT

Comparing floats with `==`. Day 2 showed that floats are close, not exact, so a correct function can fail a test:

```

>>> 0.1 + 0.2 == 0.3
False
>>> round(0.1 + 0.2, 2) == 0.3
True

```

When a result is a float, round it before comparing, as in `assert round(total, 2) == 0.3`.
pytest also offers `pytest.approx`, which means "equal, give or take a tiny amount":
`assert total == pytest.approx(0.3)`, in a test file that has `import pytest` at the top.

PAUSE AND TRY

Question 6 Multiple choice

`smallest` should return the smallest number in a list. Which call reveals its bug?

`smallest_bug.py`

```
1 def smallest(numbers: list[int]) -> int:
2     """Return the smallest number in a list that is not empty."""
3     best = 100
4     for number in numbers:
5         if number < best:
6             best = number
7     return best
8
9
10 print(smallest([40, 7, 12]))
11 print(smallest([7]))
12 print(smallest([-3, -8]))
13 print(smallest([150, 120]))
```

- A. `smallest([40, 7, 12])`
- B. `smallest([7])`
- C. `smallest([-3, -8])`
- D. `smallest([150, 120])`

Question 7 Short answer

`count_vowels(text)` should return how many of a, e, i, o, u appear in `text`, in upper or lower case. List five test cases (input and expected result), and say what kind of case each one is.

20.5 Testing Expected Exceptions

Refusing bad input is part of what a function does. `BankAccount` (Days 16 and 17) must refuse to withdraw more than its balance by raising `ValueError`, and a test should prove it does. If a later change lets an 80-dollar withdrawal through from a 50-dollar account, you want a test to fail.

Here is `BankAccount` again, with type hints and docstrings in Day 19's style:

bank/bank.py

```
1 """The BankAccount class from Days 16 and 17, with type hints."""
2
3
4 class BankAccount:
5     """An account with an owner and a balance that cannot go negative."""
6
7     def __init__(self, owner: str, balance: float = 0) -> None:
8         if balance < 0:
9             raise ValueError(f"opening balance cannot be negative, not {balance}")
10        self.owner = owner
11        self.balance = balance
12
13    def deposit(self, amount: float) -> None:
14        """Add amount to the balance.
15
16        Raises:
17            ValueError: If amount is 0 or less.
18        """
19        if amount <= 0:
20            raise ValueError(f"deposit must be more than 0, not {amount}")
21        self.balance += amount
22
23    def withdraw(self, amount: float) -> None:
24        """Take amount out of the balance.
25
26        Raises:
27            ValueError: If amount is 0 or less, or more than the balance.
28        """
29        if amount <= 0:
30            raise ValueError(f"withdrawal must be more than 0, not {amount}")
31        if amount > self.balance:
32            raise ValueError(f"only {self.balance:.2f} available, not {amount:.2f}")
33        self.balance -= amount
34
35    def __str__(self) -> str:
36        return f"{self.owner}'s account: {self.balance:.2f}"
```

A test cannot just call `account.withdraw(80)`: when the exception is raised, it would crash the test. The test has to **expect** the exception: pass if it is raised, and fail if it is not. With only Day 15's tools, that takes a `try` statement:

bank/raises_by_hand.py

```

1  from bank import BankAccount
2
3
4  def test_withdraw_more_than_balance_is_refused() -> None:
5      account = BankAccount("Ana", 50)
6      try:
7          account.withdraw(80)
8      except ValueError:
9          pass
10     else:
11         raise AssertionError("withdraw(80) did not raise ValueError")
12     assert account.balance == 50
13
14
15 test_withdraw_more_than_balance_is_refused()
16 print("Test passed: withdraw(80) was refused, and the balance is still 50")

```

Test passed: withdraw(80) was refused, and the balance is still 50

If `withdraw(80)` raises `ValueError`, the `except` clause catches it and does nothing (`pass`), and the test carries on. If it raises nothing, the `else` clause (Day 15) runs and raises `AssertionError` itself, which fails the test. Line 12 then checks that the refused withdrawal left the balance alone.

That is a lot of lines for one idea, so `pytest` provides `pytest.raises`, used in a `with` statement (Day 14):

```

1  import pytest
2
3  from bank import BankAccount
4
5
6  def test_withdraw_more_than_balance_is_refused() -> None:
7      account = BankAccount("Ana", 50)
8      with pytest.raises(ValueError):
9          account.withdraw(80)
10     assert account.balance == 50
11
12
13 def test_withdraw_message_says_what_is_available() -> None:
14     account = BankAccount("Ana", 50)
15     with pytest.raises(ValueError, match="only 50.00 available"):
16         account.withdraw(80)

```

- `import pytest` comes first, in its own group of third-party imports, then the import of your own module (Day 19's import order).
- `with pytest.raises(ValueError):` means "the code in this block must raise a `ValueError`". If it does, `pytest` catches it and the test carries on after the block. If the block finishes without raising, the test fails with `Failed: DID NOT RAISE ValueError`. If the block raises a **different** exception, such as `TypeError`, that exception is not caught, and the test fails with it.

- `match="..."` also checks the message: the text must appear in the exception's message. A few characters, such as `.` and `(`, have special meanings in `match`; plain words and numbers work as you would expect.

KEY IDEA

Raising an exception for bad input is behavior, so test it. `with pytest.raises(ValueError):` passes only if its block raises that exception, and `match=` also checks the message. Then check that the failed action changed nothing.

WATCH OUT

Putting more than the raising line inside the block. The block ends the moment the exception is raised, so any line after it is skipped:

```
1 with pytest.raises(ValueError):
2     account.withdraw(80)
3     assert account.balance == 50 # never runs
```

The balance check looks like part of the test, but it can never fail. Put only the line that should raise inside the `with` block, and the checks after it, one indent level out.

PAUSE AND TRY

Question 8 Multiple choice

Which test correctly checks that `deposit(0)` is refused? Each one starts with `account = BankAccount("Ana", 50)`.

- A. `with pytest.raises(ValueError):`, then `account.deposit(0)` indented inside it
- B. `account.deposit(0)`, then `assert ValueError` on the next line
- C. `try: with account.deposit(0) inside, then except ValueError: pass`
- D. `with pytest.raises(ValueError): pass`, then `account.deposit(0)` after the block

20.6 Finding a Bug Systematically

A failing test tells you **that** something is wrong and which input shows it. **Debugging** is the work of finding **why**: locating the line that causes the wrong behavior, called the **bug**, and fixing it. Beginners often debug by guessing: change a `>` to `>=`, run, change it back, try something else. That can take an hour, and even when the output looks right, you do not know why. A systematic approach finds the cause on purpose:

1. **Reproduce it.** Find an input that shows the bug every time, and write it down as a failing test.
2. **Say what should happen.** Before looking at the code, work out by hand what each variable should hold at each step.
3. **Narrow it down.** Look at what the variables really hold, and find the **first** place where they differ from what you expected. The bug is at or just before that point.

4. **Fix the cause**, not the symptom. Then run **every** test, and keep the new one, so this bug can never come back unnoticed.

Step 3 needs a way to see inside a running program. The simplest is the one you already know: `print`.

WORKED EXAMPLE Watching a loop with print

`longest_streak` should return the most days in a row on which a step count reached the goal. It works for the first week below, and gives the wrong answer for the second:

`streak.py`

```
1 def longest_streak(steps: list[int], goal: int) -> int:
2     """Return the most days in a row on which steps reached the goal."""
3     best = 0
4     current = 0
5     for count in steps:
6         if count >= goal:
7             current += 1
8         else:
9             if current > best:
10                best = current
11                current = 0
12     return best
13
14
15 print(longest_streak([4000, 12000, 11000, 3000], 10000))
16 print(longest_streak([12000, 4000, 11000, 10500, 13000], 10000))
```

```
2
1
```

The second week has 12000, then 4000, then three days in a row over 10000, so the answer should be 3. To see what the loop is doing, add one line at the end of the loop body:

`streak_print.py`

```
1 def longest_streak(steps: list[int], goal: int) -> int:
2     """Return the most days in a row on which steps reached the goal."""
3     best = 0
4     current = 0
5     for count in steps:
6         if count >= goal:
7             current += 1
8         else:
9             if current > best:
10                best = current
11                current = 0
12         print(f"{count=} {current=} {best=}")
13     return best
14
15
16 print(longest_streak([12000, 4000, 11000, 10500, 13000], 10000))
```

```

count=12000 current=1 best=0
count=4000 current=0 best=1
count=11000 current=1 best=1
count=10500 current=2 best=1
count=13000 current=3 best=1
1

```

Line 12 uses a shortcut made for debugging: inside an f-string's braces, a name followed by `=`, as in `f"{count=}"`, prints the name, an equals sign, and the value. Now compare each line with what you expected. `current` counts correctly, reaching 3 on the last day. But `best` is updated only on line 10, inside the `else` branch, which runs when a day **misses** the goal. The last streak never meets a missed day, so `best` never hears about it, and line 13 returns 1. The first week passed only because it ended with a missed day. The fix: after the loop, compare the final streak once more.

`streak_fixed.py`

```

1 def longest_streak(steps: list[int], goal: int) -> int:
2     """Return the most days in a row on which steps reached the goal."""
3     best = 0
4     current = 0
5     for count in steps:
6         if count >= goal:
7             current += 1
8         else:
9             if current > best:
10                best = current
11                current = 0
12     if current > best:
13         best = current
14     return best
15
16
17 print(longest_streak([4000, 12000, 11000, 3000], 10000))
18 print(longest_streak([12000, 4000, 11000, 10500, 13000], 10000))

```

```

2
3

```

Then delete the debugging `print`. It did its job, and left in, it would clutter the program's output for every user. (20.9 shows a tool for messages you want to keep.)

KEY IDEA

Debug systematically: reproduce the bug with a test, work out what the values should be, inspect what they really are to find the first place they differ, then fix the cause and rerun every test. `print(f"{name=}")` shows a variable's name and value.

PAUSE AND TRY**Question 9** Tracing

`count_long_words` should count the words with **at least** `length` letters. What does this program print, and what does the debugging output tell you about the bug?

`debug_trace.py`

```
1 def count_long_words(text: str, length: int) -> int:
2     """Return how many words in text have at least length letters."""
3     count = 0
4     for word in text.split():
5         if len(word) > length:
6             count += 1
7             print(f"{word=} {count=}")
8     return count
9
10
11 print(count_long_words("tests catch every small bug", 5))
```

Question 10 Multiple choice

A classmate reports that your `average` function gives a wrong answer "sometimes". What should you do first?

- A. Rewrite the function in a different way and hope the bug goes away
- B. Find an input that gives the wrong answer every time, and write it as a failing test
- C. Add a `print` to every line of the program
- D. Change each `<` to `<=` one at a time until the answer looks right

20.7 Pausing a Program with `breakpoint()`

`print` has limits. You have to decide in advance which values to show, edit the code, run it, read the output, and then edit again to show something else. And every `print` must be removed afterward. A **debugger** is a tool that pauses a running program at a line you choose, so you can look at any variable, run the program one line at a time, and watch the values change. Python includes one, called **pdb** (the Python debugger), which runs in the terminal.

The easiest way to start it is the built-in function `breakpoint()`. A **breakpoint** is a place where the program pauses. Put the call on its own line where you want to pause, and run the program as usual:

```

1 def longest_streak(steps: list[int], goal: int) -> int:
2     """Return the most days in a row on which steps reached the goal."""
3     best = 0
4     current = 0
5     breakpoint()
6     for count in steps:
7         if count >= goal:
8             current += 1
9         else:
10            if current > best:
11                best = current
12            current = 0
13    return best
14
15
16 print(longest_streak([12000, 4000, 11000, 10500, 13000], 10000))

```

When Python reaches line 5, it stops and gives you the debugger's prompt, (Pdb), where you type commands instead of Python code. These are the ones you need:

Command	Short	What it does
p name		Print a variable, or any expression, such as p count, best
next	n	Run the current line, and stop at the next line in this function
step	s	Like n, but if the line calls a function, stop at that function's first line
continue	c	Run until the next breakpoint, or until the program ends
break 13	b 13	Set a breakpoint at line 13 without editing the file
list	l	Show the lines around the current one
where	w	Show the chain of calls that led here, like a traceback
quit	q	Stop the program

Running one line at a time like this is called **stepping** through the code. Here is a session (sample output; the > lines show the file, the line number, and the function, and the -> line is the line about to run, **not yet** run):

```

> .../day20/streak_debug.py(5)longest_streak()
-> breakpoint()
(Pdb) n
> .../day20/streak_debug.py(6)longest_streak()
-> for count in steps:
(Pdb) n
> .../day20/streak_debug.py(7)longest_streak()
-> if count >= goal:
(Pdb) p count
12000
(Pdb) n
> .../day20/streak_debug.py(8)longest_streak()
-> current += 1
(Pdb) n
> .../day20/streak_debug.py(6)longest_streak()
-> for count in steps:
(Pdb) p current, best
(1, 0)
(Pdb) b 13
Breakpoint 1 at .../day20/streak_debug.py:13
(Pdb) c
> .../day20/streak_debug.py(13)longest_streak()
-> return best
(Pdb) p count, current, best
(13000, 3, 1)
(Pdb) q
Quitting pdb will kill the process. Quit anyway? [y/n] y

```

Step by step: `n` moves to the `for` line, then into the loop, where `p count` shows the first day's 12000. Two more `n` commands run `current += 1` and go back to the `for` line, and `p current, best` confirms `(1, 0)`. Rather than step through every day, `b 13` sets a breakpoint on `return best`, and `c` runs the rest of the loop up to it. There, `p count, current, best` shows `(13000, 3, 1)`: the loop is over, `current` is 3, `best` is 1, and the function is about to return the 1. The same discovery as with `print`, without changing what the program prints. Older versions of Python may not ask before quitting, and they may first stop on the line **after** `breakpoint()` instead of on it.

You can also start the debugger without editing the file: `python -m pdb streak.py` pauses before the first line, and `b longest_streak` then `c` runs to the start of that function.

KEY IDEA

`breakpoint()` pauses the program and opens `pdb` at the `(Pdb)` prompt. `p` shows values, `n` runs one line, `s` steps into a call, `b` sets another breakpoint, `c` continues to it, and `q` quits. The `->` line has not run yet.

WATCH OUT

Typing a variable's name when it is also a command. If a function has a variable called `n`, `c`, `s`, `l`, or `q`, typing that name at `(Pdb)` runs the command instead: in `is_prime(n)`, typing `n` steps to the next line rather than showing the number. Always use `p`, as in `p n`. And when you

have finished, delete the `breakpoint()` line. Left in, it pauses the program for whoever runs it next, waiting for debugger commands they do not know.

PAUSE AND TRY

Question 11 Multiple choice

`pdb` has stopped on the line `total = add_tax(price)`, and you suspect the bug is inside `add_tax`. Which command takes you to the first line of `add_tax`?

- A. `n`
- B. `s`
- C. `c`
- D. `p add_tax(price)`

20.8 Stepping Through Code in Your Editor

`pdb` works anywhere Python runs, which matters on a computer with no editor. When you have one, your editor's **debugger** does the same job with buttons and panels, and shows every variable at once instead of one `p` at a time. Here is how it works in VS Code with the Python extension you installed on Day 1; other editors, such as PyCharm, have the same parts under similar names, and menu labels change a little between versions.

1. **Set a breakpoint.** Click in the margin just left of a line number. A red dot appears. No code changes, so there is nothing to delete later.
2. **Start debugging.** Open the Run and Debug view and choose **Run and Debug** (or press F5), then choose the Python debugger and **Python File**. The program runs and pauses at the red dot, with the line about to run highlighted.
3. **Inspect.** The **Variables** panel lists every local variable and its current value, and updates after each step. Hover over a name in the code to see its value. Add an expression such as `current > best` to the **Watch** panel to see it recalculated at every step. The **Call Stack** panel lists the calls that led here, the same chain a traceback shows.
4. **Step.** A small toolbar appears:

Button	Key	Same as <code>pdb</code>	What it does
Continue	F5	<code>c</code>	Run to the next breakpoint
Step Over	F10	<code>n</code>	Run the current line, including any calls in it, and stop
Step Into	F11	<code>s</code>	If the line calls your function, stop at that function's first line
Step Out	Shift+F11	(none here)	Finish the current function and stop back in its caller
Stop	Shift+F5	<code>q</code>	End the program

1. **Use a condition when the loop is long.** Right-click a red dot and add a condition, such as `count < goal`. The program then pauses on that line only when the condition is true, so you can skip straight to the case that matters, even in a loop of ten thousand items.

WORKED EXAMPLE Finding the streak bug in the editor

In `streak.py`, click left of line 6 (`if count >= goal:`) to set a breakpoint, and start debugging. The first call, for the week that works, pauses four times; press F5 (Continue) through those. The fifth pause is the second call's first day, with `count` at 12000 in the Variables panel. Add `best` and `current` to Watch. Press F5 once per day: at each pause the panel shows `current` going 1, 0, 1, 2, and `best` going 0, 1, 1, 1. From the last day's pause, press F10 (Step Over) a few times: `current` becomes 3, the loop ends, and the highlight lands on `return best` with `best` still 1. The editor shows the same facts as `streak_print.py`, without a single edit to the code.

KEY IDEA

An editor's debugger pauses at red-dot breakpoints and shows every variable live. Step Over runs a line, Step Into enters a call, Step Out finishes the current function, and a conditional breakpoint pauses only when its condition is true.

PAUSE AND TRY**Question 12** Multiple choice

The debugger is paused on the first line of a function you called by mistake while stepping. You do not care what happens inside it. Which button gets you back to the line that called it, with the call finished?

- A. Step Over
- B. Step Into
- C. Step Out
- D. Stop

20.9 A First Look at logging

Some programs cannot be paused: a script that runs every night, or a program a friend runs on another computer. The only way to know what happened is for the program to report as it goes. Debugging prints could do that, but they are all or nothing: you delete them, or every user sees them. The standard library's `logging` module lets a program write messages that you can keep in the code and switch on or off from one place. Each message is a **log message**, and each has a **log level**, which says how important it is:

Level	Function	Use it for
DEBUG	<code>logging.debug()</code>	Details only useful while hunting a bug, such as a loop's values
INFO	<code>logging.info()</code>	Normal progress: "Copied 120 files"
WARNING	<code>logging.warning()</code>	Something unexpected that the program can carry on after
ERROR	<code>logging.error()</code>	Something failed: a file could not be copied
CRITICAL	<code>logging.critical()</code>	The program cannot continue at all

Every program has two output channels. **Standard output** (`stdout`) is where `print` writes. **Standard error** (`stderr`) is a second channel for problems, where tracebacks go. A terminal shows

both, mixed together, so you rarely notice the difference. But the `>` from Day 13 sends only standard output into a file, and errors still appear on the screen, where you will see them. Logging writes to standard error unless you say otherwise. `logging.basicConfig()` sets up logging, once, at the start of a program; its `stream=sys.stdout` argument makes messages go to standard output instead, which today's examples use so the messages appear in the same channel as `print`. (`sys.stdout` is standard output, from Day 13's `sys` module.)

WORKED EXAMPLE Levels and the default setup

log_levels.py

```
1 import logging
2 import sys
3
4 logging.basicConfig(stream=sys.stdout)
5
6 print("Starting the backup")
7 logging.debug("Looking in the photos folder")
8 logging.info("Copied 120 files")
9 logging.warning("The disk is 90% full")
10 logging.error("Could not copy holiday.jpg")
11 print("Backup finished")
```

```
Starting the backup
WARNING:root:The disk is 90% full
ERROR:root:Could not copy holiday.jpg
Backup finished
```

Two of the four messages are missing. By default, logging shows only `WARNING` and above, so the `debug` and `info` calls on lines 7 and 8 print nothing. The ones that appear use the default format: the level, a colon, `root` (the name of the standard logger, which you can ignore today), another colon, and the message.

To see everything, and in a tidier format, pass two more arguments to `basicConfig`: `level=` sets the lowest level to show, and `format=` sets the layout. In the format string, `%(levelname)s` stands for the level and `%(message)s` for the message. Here is `longest_streak`, fixed, with its debugging `print` replaced by a log message, and a warning for a week with no good days:

log_streak.py

```
1 import logging
2 import sys
3
4 logging.basicConfig(
5     level=logging.DEBUG,
6     format="%(levelname)s: %(message)s",
7     stream=sys.stdout,
8 )
9
10
11 def longest_streak(steps: list[int], goal: int) -> int:
12     """Return the most days in a row on which steps reached the goal."""
13     best = 0
14     current = 0
```

```
15     for count in steps:
16         if count >= goal:
17             current += 1
18         else:
19             if current > best:
20                 best = current
21                 current = 0
22             logging.debug(f"{count=} {current=} {best=}")
23     if current > best:
24         best = current
25     if best == 0:
26         logging.warning(f"No day reached the goal of {goal}")
27     return best
28
29
30 print(longest_streak([12000, 4000, 11000], 10000))
31 print(longest_streak([3000, 4000], 10000))
```

```
DEBUG: count=12000 current=1 best=0
DEBUG: count=4000 current=0 best=1
DEBUG: count=11000 current=1 best=1
1
DEBUG: count=3000 current=0 best=0
DEBUG: count=4000 current=0 best=0
WARNING: No day reached the goal of 10000
0
```

Now change line 5 to `level=logging.WARNING` and run it again:

```
1
WARNING: No day reached the goal of 10000
0
```

The debug messages are gone, but line 22 is still in the code, ready for the next time you need it: change one word back to `DEBUG`. That is the difference from `print`. Real programs also add the time to each message, with `%(asctime)s` in the format, and can send messages to a file instead of the screen; the Go further link shows how.

KEY IDEA

`logging.debug`, `info`, `warning`, `error`, and `critical` write messages with levels. `basicConfig` with `level=` and `format=` sets, in one place, which levels appear and how. Messages go to standard error unless you pass `stream=sys.stdout`.

WATCH OUT

Expecting debug messages without setting the level. This program sets up logging but not its level:

log_missing.py

```
1 import logging
2 import sys
3
4 logging.basicConfig(stream=sys.stdout)
5
6 total = 0
7 for price in [2.5, 4.0, 1.25]:
8     total += price
9     logging.debug(f"{price=} {total=}")
10 logging.info("Loop finished")
11 print(f"Total: {total:.2f}")
```

Total: 7.75

Neither the `debug` nor the `info` message appears, because the default level is `WARNING`. Add `level=logging.DEBUG` to `basicConfig`. Also call `basicConfig` before the first log message: the first message sets up logging with the defaults if nothing has yet, and after that `basicConfig` quietly does nothing.

PAUSE AND TRY

Question 13 Tracing

What does this program print?

log_trace.py

```
1 import logging
2 import sys
3
4 logging.basicConfig(
5     level=logging.INFO,
6     format="%(levelname)s: %(message)s",
7     stream=sys.stdout,
8 )
9
10 scores = [72, 105, 88]
11 logging.debug(f"Checking {len(scores)} scores")
12 for score in scores:
13     if score > 100:
14         logging.warning(f"Skipping {score}: more than 100")
15     else:
16         logging.info(f"Score {score} is fine")
17 print("Checked")
```

Putting it together

A running club stores race times as text, such as "4:05" for 4 minutes 5 seconds. Here is a module that converts them and finds the fastest, written with tests, edge cases, expected exceptions, and logging in mind:

race/race.py

```

1  """Race times written as minutes and seconds, such as "4:05"."""
2
3  import logging
4  import sys
5
6
7  def parse_time(text: str) -> int:
8      """Return the number of seconds in a time written as m:ss.
9
10     Raises:
11         ValueError: If text is not whole minutes, a colon, and two
12             digits of seconds from 00 to 59.
13     """
14     parts = text.split(":")
15     if len(parts) != 2 or len(parts[1]) != 2:
16         raise ValueError(f"time must look like 4:05, not {text}")
17     minutes = int(parts[0])
18     seconds = int(parts[1])
19     if minutes < 0:
20         raise ValueError(f"minutes cannot be negative, not {minutes}")
21     if not 0 <= seconds <= 59:
22         raise ValueError(f"seconds must be 00 to 59, not {parts[1]}")
23     total = minutes * 60 + seconds
24     logging.debug(f"{text} is {total} seconds")
25     return total
26
27
28  def fastest(times: list[str]) -> str:
29      """Return the fastest time in a list.
30
31     Raises:
32         ValueError: If times is empty.
33     """
34     return min(times, key=parse_time)
35
36
37  if __name__ == "__main__":
38     logging.basicConfig(
39         level=logging.DEBUG,
40         format="%(levelname)s: %(message)s",
41         stream=sys.stdout,
42     )
43     print("Fastest:", fastest(["10:00", "9:59", "12:01"]))
44     try:
45         parse_time("3:60")
46     except ValueError as error:
47         print("Refused:", error)

```

Running the module directly turns on debug logging and tries it out:

```
DEBUG: 10:00 is 600 seconds
DEBUG: 9:59 is 599 seconds
DEBUG: 12:01 is 721 seconds
Fastest: 9:59
Refused: seconds must be 00 to 59, not 60
```

And here is its test file, `test_race.py`, for pytest:

```
1 import pytest
2
3 from race import fastest, parse_time
4
5
6 def test_parse_typical_time() -> None:
7     assert parse_time("4:05") == 245
8
9
10 def test_parse_smallest_and_largest_seconds() -> None:
11     assert parse_time("0:00") == 0
12     assert parse_time("3:59") == 239
13
14
15 def test_parse_two_digit_minutes() -> None:
16     assert parse_time("12:30") == 750
17
18
19 def test_parse_rejects_60_seconds() -> None:
20     with pytest.raises(ValueError, match="seconds must be 00 to 59"):
21         parse_time("3:60")
22
23
24 def test_parse_rejects_wrong_format() -> None:
25     with pytest.raises(ValueError):
26         parse_time("4.05")
27     with pytest.raises(ValueError):
28         parse_time("4:5")
29
30
31 def test_fastest_compares_times_not_text() -> None:
32     assert fastest(["10:00", "9:59", "12:01"]) == "9:59"
33
34
35 def test_fastest_of_one_time() -> None:
36     assert fastest(["5:00"]) == "5:00"
37
38
39 def test_fastest_of_empty_list_is_refused() -> None:
40     with pytest.raises(ValueError):
41         fastest([])
```

`python -m pytest` in the `race` folder reports (sample output):

```
collected 8 items

test_race.py ..... [100%]

===== 8 passed in 0.02s =====
```

Where	What to notice	Idea
race.py, lines 15–22	Bad input raises <code>ValueError</code> with a clear message, never an <code>assert</code>	20.2, 20.5
race.py, line 24	A debug message that stays in the code and is silent unless a program turns it on	20.9
race.py, lines 37–42	Logging is set up only when the file is run, so importing it (as the tests do) configures nothing	20.9
test_parse_typical_time	A normal case: $4 \times 60 + 5 = 245$	20.4
test_parse_smallest_and_largest_seconds	Boundaries: 00 and 59 are the smallest and largest allowed seconds	20.4
test_parse_rejects_60_seconds	The other side of the boundary, with <code>match=</code> checking the message	20.5
test_fastest_compares_times_not_text	As text, "10:00" comes before "9:59" because "1" comes before "9"; <code>min</code> with <code>key=parse_time</code> (Day 10) compares seconds	20.4
test_fastest_of_empty_list_is_refused	<code>min([])</code> raises <code>ValueError</code> , as the docstring promises	20.5

The test names read like a list of the module's promises. If someone later "simplifies" `fastest` to `min(times)`, `test_fastest_compares_times_not_text` fails at once and says exactly which promise broke. When "3:60" is refused, the exception comes from line 22: `int("60")` is fine, but `not 0 <= 60 <= 59` is true. And "4.05" has no colon, so `split(":")` gives a one-item list, and line 16 raises.

Common mistakes

Use this table to go from what you see to its usual cause. Messages from `pytest` are samples; their wording varies by version.

What you see	From	Usual cause	Fix
<code>ModuleNotFoundError: No module named 'pytest'</code>	Python	<code>pytest</code> is not installed in the active environment	Activate the environment, <code>python -m pip install</code> then <code>pytest</code>
collected 0 items, no tests ran	pytest	File or function names do not start with <code>test</code>	Rename them <code>test_*.py</code> and <code>test_*</code>
Fewer items collected than you wrote	pytest	One test function is misnamed, or two have the same name	Rename it

Failed: DID NOT RAISE ValueError	pytest	The code in a <code>pytest.raises</code> block did not raise	Fix the code, or move the raising line into the block
A bare <code>AssertionError</code>	Python	A plain <code>assert</code> without a message failed, outside <code>pytest</code>	Add a message, or run it under <code>pytest</code>
<code>SyntaxWarning: assertion is always true</code>	Python	<code>assert (condition, message)</code> with parentheses	Remove the parentheses
A float test fails by a tiny amount	Python	<code>==</code> on floats	<code>round()</code> both sides, or <code>pytest.approx</code>
<code>*** NameError at (Pdb)</code>	<code>pdb</code>	The variable has not been assigned yet at this line	Step with <code>n</code> until it has
Debug messages do not appear	<code>logging</code>	The level is still the default, <code>WARNING</code>	Pass <code>level=logging.DEBUG</code> to <code>basicConfig</code>

Vocabulary

Term	Definition	Example
automated test	Code that runs other code on known inputs and reports any result that does not match	<code>test_senior_price()</code>
test case	One input together with its expected result	65 → 7
regression	A bug that breaks something that used to work	A change that makes seniors pay 10
assert statement	Raises <code>AssertionError</code> if its condition is false; does nothing if it is true	<code>assert price == 7, "seniors pay 7"</code>
AssertionError	The exception raised when an <code>assert</code> fails	<code>AssertionError: discount(100) gave 100</code>
pytest	A third-party tool that finds and runs tests and reports the results	<code>python -m pytest</code>
test discovery	How <code>pytest</code> finds tests: files named <code>test_*.py</code> , functions named <code>test_*</code>	<code>test_tickets.py</code>
test function	A function, named <code>test_...</code> , that checks one behavior with <code>assert</code>	<code>def test_one_item() -> None:</code>
normal case	An ordinary, everyday input	<code>[72, 88, 95]</code>
edge case	An input at the edge of what a function handles, such as empty, zero, one item, or a boundary	<code>[], [-5, -2]</code>
pytest.raises	A <code>with</code> block that passes only if its code raises the given exception	<code>with pytest.raises(ValueError):</code>
debugging	Finding and fixing the cause of wrong behavior	Reproduce, inspect, fix, retest

bug	A mistake in code that makes it behave wrongly	<code>></code> where the rule needs <code>>=</code>
debugger	A tool that pauses a running program so you can inspect variables and run one line at a time	<code>pdb</code> , the VS Code debugger
pdb	Python's built-in debugger, used at the <code>(Pdb)</code> prompt	<code>python -m pdb streak.py</code>
breakpoint	A place where the debugger pauses the program	<code>breakpoint()</code> , a red dot
stepping	Running a program one line at a time in a debugger	<code>n</code> , Step Over
logging	The standard library module for writing log messages with levels	<code>logging.warning("Disk full")</code>
log level	How important a log message is: <code>DEBUG</code> , <code>INFO</code> , <code>WARNING</code> , <code>ERROR</code> , or <code>CRITICAL</code>	<code>level=logging.DEBUG</code>
standard output	The output channel <code>print</code> writes to (<code>stdout</code>)	<code>sys.stdout</code>
standard error	A second output channel for problems (<code>stderr</code>), where tracebacks and log messages go by default	A traceback

Summary

- An **automated test** checks your code against test cases whose answers come from the rule, and reruns in seconds, so it catches **regressions** the moment a change breaks something. `assert condition`, `message` raises `AssertionError` when a check fails; use it for checking code, never for validating input.
- **pytest** finds files named `test_*.py` and functions named `test_*`, runs them all with `python -m pytest`, and for each failure shows the line and the values on each side of the comparison.
- Choose test cases that reveal mistakes: a normal case, then **edge cases** such as each side of a boundary, empty or single items, and negative values. Test refusals too: `with pytest.raises(ValueError):` passes only if its block raises.
- **Debug** systematically: reproduce the bug with a test, work out what the values should be, find the first place they differ, then fix the cause and rerun every test. `print(f"{name=}")` shows values quickly.
- A **debugger** (`pdb` through `breakpoint()`, or your editor's) pauses at a **breakpoint** so you can inspect variables and step line by line. **logging** writes messages with levels that you switch on or off in one place with `basicConfig`.

Practice

Do these in your `day20` folder, with `pytest` installed in your virtual environment (20.3). Type each program yourself, run it, and only then open the answer.

Question 14 Short answer

Testing `is_prime`. Copy `is_prime` from Day 9 into `primes.py`, adding type hints and a docstring. Then write five tests for it in `test_primes.py`, including the edge cases 0, 1, and 2, and run them with `pytest`.

Question 15 Short answer

Testing `BankAccount`. Save `bank.py` from 20.5 in your `day20` folder. Write `test_bank.py` with tests for creating an account, depositing, and withdrawing, including the edge case of withdrawing the whole balance, and at least one test that expects an exception.

Question 16 Short answer

Find a planted bug with the debugger. Plant a bug in one of your own functions, run it, and find the bug with `pdb` or your editor's debugger, without using `print`. If you would like a bug to practice on first, this function should average a list of scores, so `[90, 80, 70]` should give 80.0:

`class_average.py`

```
1 def class_average(scores: list[int]) -> float:
2     """Return the average of scores, which must not be empty."""
3     total = 0
4     for index in range(1, len(scores)):
5         total += scores[index]
6     return total / len(scores)
7
8
9 print(class_average([90, 80, 70]))
```

Review questions

Question 17 Multiple choice

`test_scores.py` contains four functions: `test_average` (passes), `test_empty_list` (fails), `check_highest` (would fail), and `test_lowest` (passes). What is the last line of the pytest report?

- A. 2 failed, 2 passed
- B. 1 failed, 2 passed
- C. 1 failed, 3 passed
- D. 1 failed, because pytest stops at the first failure

Question 18 Debugging

This test passes. Then someone deletes the balance check from `withdraw`, so withdrawing 80 from 50 leaves -30, and the test **still** passes. Explain why, and fix the test.

```
1 def test_withdraw_too_much() -> None:
2     account = BankAccount("Ana", 50)
3     with pytest.raises(Exception):
4         account.withdraw(80)
```

Question 19 Short answer

`letter_grade(score)` returns "A" for 90 to 100, "B" for 80 to 89, "C" for 70 to 79, "D" for 60 to 69, and "F" below 60, and raises `ValueError` for a score below 0 or above 100. Plan its tests: list the inputs you would test, with the expected result for each, and explain your choices.

Question 20 Short answer

A classmate checks the user's input like this:

```
1 age = int(input("Age: "))
2 assert age >= 0, "age cannot be negative"
```

Give two reasons this is a poor way to refuse bad input, and say what to write instead.

Question 21 Multiple choice

A program processes 5,000 orders and gives a wrong total. You have found that the problem appears only for the order whose `order_id` is "A-4812". What is the quickest way to inspect the variables at that moment?

- A. Add `print(f"{order=}")` inside the loop and read all 5,000 lines
- B. Set a conditional breakpoint inside the loop with the condition `order_id == "A-4812"`
- C. Set an ordinary breakpoint inside the loop and press Continue until you reach it
- D. Add `logging.debug(f"{order=}")` with the level left at its default

Question 22 Short answer

All twelve of your tests pass. A classmate says that proves your function has no bugs. Are they right? Explain what passing tests do and do not prove, and how to make them prove more.

Go further

The pytest documentation's Get Started page walks through installing pytest, writing a first test, and testing exceptions, with a little more than today covers. When you want to keep logs with times, or write them to a file, the Logging HOWTO in the Python documentation is the place to go next.

Reflection

Check each statement. If the answer is "not yet," reread the linked subsection.

- ☐ I can explain why automated tests catch regressions, and what makes a good test case.

Reread: 20.1 Why Automated Tests

- ☐ I can write an assert with a message, and explain why assert is not for checking user input.

Reread: 20.2 Checking with assert

- ☐ I can install pytest, name test files and functions so it finds them, and read its report. Reread: 20.3 Running Tests with pytest

- ☐ I can choose normal cases and edge cases that are likely to reveal a bug. Reread: 20.4 Normal Cases and Edge Cases

- ☐ I can test that code raises an exception with `pytest.raises`. Reread: 20.5 Testing Expected Exceptions

- ☐ I can find a bug systematically, and use print to inspect values. Reread: 20.6 Finding a Bug Systematically

- ☐ I can pause a program with `breakpoint()` and use p, n, s, b, c, and q at the (Pdb) prompt.

Reread: 20.7 Pausing a Program with `breakpoint()`

- ☐ I can set breakpoints in my editor, step over, into, and out, and watch variables. Reread: 20.8 Stepping Through Code in Your Editor

☐ I can write log messages at different levels and choose which ones appear. Reread: 20.9 A First Look at logging