Module 18 of 20

Testing and debugging

The testing mindset and designing for testability; unittest with its assertion family, fixtures, subTest and in-process runs; pytest's fixtures and parametrize and the mechanisms behind them; test doubles with Mock, side_effect and patch; and debugging by reading tracebacks, logging, pdb and the reproduce–minimise–bisect method.

A test is a program that runs your code with known inputs and checks the outputs, so that a machine catches a regression before a user does. Everything in this module follows from that: tests pin behaviour, not implementation; they cover edges and errors; they are fast, isolated and deterministic; and the code they exercise is written so the world — the clock, randomness, the network, a sender — can be passed in and replaced. Debugging is the same discipline in reverse: reading what the interpreter already reports, recording the path the program took, and shrinking the mystery until the cause is one line.

The testing mindset gives the kinds of test, arrange–act–assert, what to test and the injection moves that make code testable. unittest covers `TestCase`, the assertion family, `setUp` / `tearDown`, `subTest`, skipping, and running a suite in-process to read its result. pytest in outline covers plain asserts, fixtures, `parametrize`, `raises`, markers and the command line — and rebuilds parametrisation and yield-fixtures in plain Python, since pytest is not on the judge. Mocking covers stubs, fakes, spies and mocks, the `Mock` API, `side_effect`, `patch` in its forms and the rule about where to patch, `spec` and `autospec`. Debugging covers reading tracebacks bottom-up, chained exceptions, `logging`, `pdb`, `assert` and the dev switches, and the method.

The exercises are whole programs: a test harness over a table of cases, an injected clock, a `unittest` suite that must catch an injected bug by name, a `subTest` table, a plain-Python `parametrize` and yield fixture, a spied notifier, a patched network call with a scripted `side_effect`, a logging pipeline with a counting filter, and a traceback parser. The checkpoint adds a bank-account suite against three injected bugs, a retry helper against a scripted mock, and a counting log handler.

CONTENTS

1 The testing mindset — what to test, how to arrange it, and designing for testability 13 min

2	unittest — TestCase, assertions, fixtures, subTest and running suites	15 min
3	pytest in outline — plain asserts, fixtures, parametrize and the command line	13 min
4	Mocking and test doubles — Mock, patch, side_effect and where to patch	15 min
5	Debugging — reading tracebacks, logging, pdb and the method	15 min
✓	Checkpoint — Testing and debugging	25 min

Exercises in this module compile on CPython 3.11. Reference solutions and the answer key follow each lesson.

LESSON 1 · 13 MIN

The testing mindset — what to test, how to arrange it, and designing for testability

A test is a program that runs your code with known inputs and checks the outputs, so that a change which breaks behaviour is caught by a machine instead of a user. That definition already settles most arguments: tests exist to catch regressions and to pin down what the code promises, not to prove correctness or to reach a coverage number. This lesson covers the kinds of test and where each earns its place, the arrange–act–assert shape, what is worth testing, the properties that make a suite trustworthy, and the design moves — pure functions, injected dependencies, fixed clocks and seeds — that make code testable in the first place.

Kinds of test

Kind	Scope	Speed	Count
Unit	one function or class, no I/O	microseconds to milliseconds	hundreds or thousands
Integration	two or more real parts together: code and a database, a parser and a file	milliseconds to seconds	tens
End-to-end	the whole system through its real interface: an HTTP request, a CLI invocation	seconds	a handful

The pyramid — many unit tests, fewer integration tests, few end-to-end — follows from cost: a unit test runs in isolation and tells you *which* line is wrong; an end-to-end test proves the system works but says little about where it failed and takes a thousand times longer. A regression test is any test written to reproduce a bug before fixing it; it belongs at the lowest level that reproduces the problem.

Arrange, act, assert

```python
def test_discount_applies_over_threshold():
    cart = Cart([Item("book", 40), Item("pen", 70)])     # arrange: the inputs
    total = cart.total(discount_over=100, rate=0.1)      # act: one call
    assert total == 99.0                                 # assert: one behavio
```

Every test has these three parts, in this order, ideally with a blank line between them. One behaviour per test, named for the behaviour (`test_discount_applies_over_threshold`, not `test_total_2`), so that a failure's name is already a diagnosis. Several assertions are fine when they describe one behaviour; a test that checks five unrelated things stops at the first failure and hides the other four.

What to test

Behaviour, not implementation. Assert on what the function returns or does to the world, not on which helpers it called; a test coupled to the implementation fails on every refactor and passes on every bug that keeps the same shape.

The edges. Empty input, one element, the boundary value (exactly 100 when the rule is "over 100"), negative, zero, the largest sensible size, Unicode, whitespace.

The errors. The invalid inputs raise the documented exception with a useful message; the recoverable failures are recovered.

The bug you just fixed. A regression test that failed before the fix and passes after.

Not private helpers, trivial getters, or the standard library.

Properties of a good suite

Fast — a suite that takes minutes is run rarely. Isolated — each test sets up its own state and leaves nothing behind, so order does not matter and one failure does not cascade. Deterministic — the same result every run: no dependence on the clock, randomness, network, the file system's contents, or hash order. Readable — a failing test explains itself from its name and its assertion message. Independent of each other — no test relies on another having run. A flaky test (passes sometimes) is worse than no test: it trains people to ignore red.

Designing for testability

The code that is easy to test is code whose inputs and outputs are explicit:

```
# hard to test: reads the clock, the environment and the network inside
def report():
    today = date.today()
    rows = requests.get(URL).json()
    ...

# easy to test: the world is passed in
def report(rows, today):
    ...

def main():
    print(report(fetch_rows(), date.today()))     # the impure shell, thin
```

The pattern is *functional core, imperative shell*: pure functions that compute from arguments, wrapped by a thin layer that does the I/O. Dependencies a function needs — the clock, a random generator, a sender, a store — are parameters with sensible defaults (`now=None` then `now = now or datetime.now()`), so a test passes a fixed value and the production caller passes nothing. `random.Random(seed)` instead of the module functions makes randomness reproducible. A class that takes its collaborators in `__init__` can be given fakes.

Doubles, briefly

When a dependency is slow, unavailable or nondeterministic, tests replace it with a *double*: a **stub** returns canned answers, a **fake** is a working lightweight implementation (an in-memory dict for a database), a **spy** records what was called, a **mock** does both and can assert on calls. Lesson 4 builds them with `unittest.mock`; the design above is what makes them pluggable.

Coverage and TDD

Coverage (`coverage run -m pytest`, then `coverage report`) shows which lines tests executed. Untested lines are a signal; 100 % is not a goal — a line can execute without its behaviour being checked. Test-driven development writes the failing test first, then the least code that passes, then refactors; it is a design technique as much as a testing one, because a test written first forces the interface to be usable. Neither is mandatory; both are tools.

Pitfalls

- Tests that assert on implementation details (which method was called, in what order) rather than outcomes.
- A test with no assertion (it "runs without crashing").
- Hidden dependence on the clock, randomness or test order.
- One giant test per module.
- Mocking so much that the test checks the mocks, not the code.
- Skipping the regression test because "the fix is obvious".

Key takeaways

- Tests catch regressions and pin promises; unit tests are many and fast, integration tests fewer, end-to-end tests few.
- Arrange, act, assert; one behaviour per test, named for it.
- Test behaviour, edges and errors, and every fixed bug; not private helpers.
- A trustworthy suite is fast, isolated, deterministic and readable — flaky tests are worse than none.
- Make code testable by passing the world in: pure core, thin impure shell, injected clock, seed and collaborators.

EXERCISE 1

A table of cases

Implement `parse_duration(text)` returning seconds for strings made of optional `<n>h`, `<n>m` and `<n>s` parts in that order (`1h30m`, `45s`, `2h`), raising `ValueError` for anything else. Then read lines `<input> <expected>` until EOF, where `expected` is an integer or the word `error`, and run each through the function: print `ok <input> → <got>` when the result (or `error` when it raised) matches, else `FAIL <input>: expected <expected>, got <got>`. Finish with `passed <x> of <y>`.

Input: one case per line. **Output:** one line per case, then the summary.

```
1h 3600
30m 1800
1h30m 5400
45s 45
2h 7000
x error
```

prints

```
ok 1h → 3600
ok 30m → 1800
ok 1h30m → 5400
ok 45s → 45
FAIL 2h: expected 7000, got 7200
ok x → error
passed 5 of 6
```

INPUT 1

```
1h 3600
30m 1800
1h30m 5400
45s 45
2h 7000
x error
```

OUTPUT 1

```
ok 1h → 3600
ok 30m → 1800
ok 1h30m → 5400
ok 45s → 45
FAIL 2h: expected 7000, got 7200
ok x → error
passed 5 of 6
```

REFERENCE SOLUTION

```
import re
import sys

PATTERN = re.compile(r"(?:(\d+)h)?(?:(\d+)m)?(?:(\d+)s)?")


def parse_duration(text):
    match = PATTERN.fullmatch(text)
    if match is None:
        raise ValueError(f"bad duration: {text!r}")
    hours, minutes, seconds = (int(part) if part else 0 for part in match.groups())
    return hours * 3600 + minutes * 60 + seconds


def run_case(text):
    """Return the result as a string: the seconds, or 'error' when it raised."""
    try:
        return str(parse_duration(text))
    except ValueError:
        return "error"


passed = total = 0
for line in sys.stdin:
    parts = line.split()
    if len(parts) != 2:
        continue
    text, expected = parts
    got = run_case(text)
    total += 1
    if got == expected:
        passed += 1
        print(f"ok {text} → {got}")
    else:
        print(f"FAIL {text}: expected {expected}, got {got}")

print(f"passed {passed} of {total}")
```

EXERCISE 2

Inject the clock

Write `status(due, today)` taking two `datetime.date` values and returning `overdue by <n> day(s)` when `due` is before `today`, `due today` when equal, and `due in <n> day(s)` otherwise — with `day` for 1 and `days` otherwise. Read `today` as an ISO date on the first line, then lines `<name>`

`<YYYY-MM-DD>` until EOF; print `<name>: <status>` per line, then `overdue <count>`. The date is a parameter, never `date.today()`, so the program is testable.

Input: the date, then one item per line. **Output:** one line per item, then the count.

```
2024-03-10
tax 2024-03-01
rent 2024-03-10
gym 2024-03-15
```

prints

```
tax: overdue by 9 days
rent: due today
gym: due in 5 days
overdue 1
```

INPUT 1

```
2024-03-10
tax 2024-03-01
rent 2024-03-10
gym 2024-03-15
```

OUTPUT 1

```
tax: overdue by 9 days
rent: due today
gym: due in 5 days
overdue 1
```

REFERENCE SOLUTION

```
import sys
from datetime import date


def days(n):
    return f"{n} day" if n == 1 else f"{n} days"


def status(due, today):
    delta = (due - today).days
    if delta < 0:
        return f"overdue by {days(-delta)}"
    if delta == 0:
        return "due today"
    return f"due in {days(delta)}"


today = date.fromisoformat(sys.stdin.readline().strip())
overdue = 0
for line in sys.stdin:
    parts = line.split()
    if len(parts) != 2:
        continue
    name, due = parts[0], date.fromisoformat(parts[1])
    text = status(due, today)
    if text.startswith("overdue"):
        overdue += 1
    print(f"{name}: {text}")

print("overdue", overdue)
```

QUIZ

1. What are the three parts of a well-shaped test, in order?

A. Setup, run, cleanup

B. Arrange, act, assert

C. Import, call, print

D. Mock, call, verify

2. Which is worth a unit test?

A. That `list.append` works

B. The boundary where a discount starts applying

C. A private helper's exact call sequence

D. A trivial getter

3. Why is a flaky test worse than no test?

A. It is slow

B. It trains people to ignore failures, so a real one is missed

C. It uses memory

D. It cannot be deleted

4. How do you make a function that uses the current date testable?

A. Mock `datetime` everywhere

B. Pass `today` as a parameter and let the caller supply `date.today()`

C. Freeze the system clock

D. Avoid dates

5. What does 100 % line coverage prove?

A. The code is correct

B. Every line executed under some test — not that its behaviour was checked

C. There are no bugs

D. The tests are fast

ANSWER KEY · THE TESTING MINDSET — WHAT TO TEST, HOW TO ARRANGE IT, AND DESIGNING FOR TESTABILITY

1. B Build the inputs, make one call, check one behaviour.

2. B Edges, errors and fixed bugs are where regressions live; the standard library and implementation details are not.

3. B Determinism is a property of a trustworthy suite.

4. B Functional core, imperative shell: the world is passed in.

5. B Coverage is a signal about untested lines, not a goal.

LESSON 2 · 15 MIN

unittest — TestCase, assertions, fixtures, subTest and running suites

`unittest` is the test framework in the standard library: tests are methods named `test_*` on a `TestCase` subclass, assertions are methods (`assertEqual`, `assertRaises`, …) that produce readable failure messages, `setUp` and `tearDown` build and clear state around each test, and `python -m unittest` discovers and runs them. It is more verbose than pytest (next lesson) and it is everywhere — CI images, older code bases, environments where nothing can be installed — so knowing it well is not optional. This lesson covers the class, the assertion family, fixtures at three scopes, `subTest` for tables, skipping and expected failures, and running a suite in-process so a program can report its own results.

The shape

```
import unittest

def slugify(text):
    return "-".join(text.lower().split())

class TestSlugify(unittest.TestCase):
    def test_lowercases_and_joins(self):
        self.assertEqual(slugify("Hello World"), "hello-world")

    def test_collapses_whitespace(self):
        self.assertEqual(slugify("  a   b "), "a-b")

    def test_empty(self):
        self.assertEqual(slugify(""), "")

if __name__ == "__main__":
    unittest.main()
```

Run with `python test_slugify.py` or `python -m unittest test_slugify` (or `python -m unittest` alone to discover `test*.py` files). Output: a dot per passing test, `F` for a failure (an assertion was false), `E` for an error (an unexpected exception), then a summary.

The assertion family

Method	Checks
`assertEqual(a, b)` / `assertNotEqual`	`a == b`, with a diff for strings, lists and dicts
`assertTrue(x)` / `assertFalse(x)`	truthiness — prefer a more specific assertion when one exists
`assertIs(a, b)` / `assertIsNone(x)`	identity
`assertIn(x, xs)` / `assertNotIn`	membership
`assertIsInstance(x, cls)`	type
`assertAlmostEqual(a, b, places=7)`	floats, rounded difference
`assertGreater`, `assertLess`, `assertGreaterEqual`, …	comparisons
`assertRaises(Exc)`	as a context manager: the block raises `Exc`
`assertRaisesRegex(Exc, pattern)`	and its message matches
`assertCountEqual(a, b)`	same elements in any order
`assertDictEqual`, `assertListEqual`, …	explicit, with diffs

Every method takes a final `msg=` argument added to the failure output. The specific methods matter because their messages are specific: `assertEqual([1, 2], [1, 3])` shows where the lists differ; `assertTrue([1, 2] == [1, 3])` says only "False is not true".

```
def test_rejects_empty_name(self):
    with self.assertRaises(ValueError) as ctx:
        User(name="")
    self.assertIn("name", str(ctx.exception))
```

Fixtures: setUp, tearDown and class-level

```python
class TestInventory(unittest.TestCase):
    @classmethod
    def setUpClass(cls):                    # once per class — expensive share
        cls.catalogue = load_catalogue()

    def setUp(self):                        # before EVERY test — fresh state
        self.inv = Inventory(self.catalogue)
        self.inv.add("bolt", 10)

    def tearDown(self):                     # after every test, even a failing
        self.inv.close()

    def test_remove(self):
        self.inv.remove("bolt", 4)
        self.assertEqual(self.inv.count("bolt"), 6)
```

`setUp` runs before each test method and `tearDown` after, so every test starts from the same state and no test sees another's leftovers. `setUpClass` / `tearDownClass` bracket the whole class; `setUpModule` the module. `self.addCleanup(fn)` registers a callback run after `tearDown` — the safest way to release something acquired mid-test. Temporary files belong in `tempfile.TemporaryDirectory()` opened in `setUp` and cleaned in `tearDown`.

---

subTest: tables without a method per row

```python
def test_parse_durations(self):
    cases = [("1h", 3600), ("30m", 1800), ("1h30m", 5400), ("45s", 45)]
    for text, expected in cases:
        with self.subTest(text=text):
            self.assertEqual(parse_duration(text), expected)
```

Without `subTest`, the loop stops at the first failing row; with it, every row is checked and each failure is reported with its parameters. The method still counts as one test in `testsRun`, but each failing subtest appears as its own entry in the failures list.

Skipping and expected failures

`@unittest.skip("reason")`, `@unittest.skipIf(cond, "reason")`, `@unittest.skipUnless(cond, "reason")` skip a test or a whole class;

`self.skipTest("reason")` skips from inside. `@unittest.expectedFailure` marks a known bug: the test passing becomes an "unexpected success". Skips are reported in the summary and are not failures — a suite that silently skips half its tests on CI because a dependency is missing is a common way to ship a regression.

Running in-process

```
import io, unittest

suite = unittest.defaultTestLoader.loadTestsFromTestCase(TestSlugify)
result = unittest.TextTestRunner(stream=io.StringIO(), verbosity=0).run(suite)
print("ran", result.testsRun)
print("failures", len(result.failures), "errors", len(result.errors))
print("ok", result.wasSuccessful())
for test, _traceback in result.failures:
    print("failed", test.id().rsplit(".", 1)[-1])
```

`unittest.main()` calls `sys.exit` with the result, which is right for a test file and wrong for a program that wants to keep going; loading a suite and running it with a `TextTestRunner` on a `StringIO` stream keeps the report ("Ran 3 tests in 0.001s", whose timing is not reproducible) off stdout and gives you the `TestResult` to print from. `unittest.main(argv=["prog"], exit=False)` is the shorter form when the module-level discovery is wanted. `result.failures` and `result.errors` are lists of `(test, traceback_text)`; `test.id()` is `module.Class.method`.

Organising a suite

One test module per source module (`test_slugify.py` beside or under `tests/` for `slugify.py`), one `TestCase` per class or cohesive function group, and names that read as sentences: `TestSlugify.test_collapses_whitespace`. `python -m unittest discover -s tests -p "test_*.py"` finds them; `-v` prints each name; `-k slug` runs the ones whose names match; `--failfast` stops at the first failure. Keep the tests importable without side effects — no network calls or file writes at module level — because discovery imports every test module before running anything.

Pitfalls

`assertTrue(a == b)` instead of `assertEqual(a, b)` — the failure message says nothing.

State created in `__init__` or at class level instead of `setUp`, shared between tests.

A test method not starting with `test_` — silently never run.

`assertRaises(Exception)` — too broad; name the exception.

`unittest.main()` inside a program that must continue.

Comparing floats with `assertEqual`.

Key takeaways

Tests are `test_*` methods on a `TestCase`; run with `python -m unittest` or `unittest.main()`.

Use the specific assertion (`assertEqual`, `assertIn`, `assertRaises` as a context manager, `assertAlmostEqual`) for a message that explains the failure.

`setUp` / `tearDown` give every test fresh state; `setUpClass` shares expensive state; `addCleanup` releases resources.

`subTest` checks every row of a table and reports each failure with its parameters.

Run in-process with `TextTestRunner(stream=StringIO())` and read `testsRun`, `failures`, `errors`, `wasSuccessful()` from the result.

EXERCISE 1

Catch the injected bug

The starter defines `Stack` with a bug injected according to the first input line: `none`, `pop` (pops from the wrong end), `peek` (peeks at the wrong end) or `empty` (pop on an empty stack returns `None` instead of raising). Write `TestStack(unittest.TestCase)` with exactly these tests: `test_push_then_pop` (push 1 and 2, expect `pop()` to give 2 then 1), `test_peek_does_not_remove` (push 1 and 2, expect `peek()` 2 and `len` 2), `test_pop_empty_raises` (`assertRaises(IndexError)` around `pop()` on a new stack) and `test_len_and_is_empty` (new stack: `is_empty()` and `len` 0; after `push(5)`: `len` 1 and not empty). Run the suite with a `TextTestRunner` on a `StringIO` stream and print `ran <n>`, `failures <f>`, `errors <e>`, `ok <True/False>`, then `failed <method name>` for each failure, sorted.

Input: the bug name. **Output:** four lines, then any failed test names.

```
none
```

prints

```
ran 4
failures 0
errors 0
ok True
```

INPUT 1

```
none
```

OUTPUT 1

```
ran 4
failures 0
errors 0
ok True
```

REFERENCE SOLUTION

```
import io
import unittest

BUG = input().strip()


class Stack:
    def __init__(self):
        self._items = []

    def push(self, item):
        self._items.append(item)

    def pop(self):
        if not self._items:
            if BUG == "empty":
                return None
            raise IndexError("pop from empty stack")
        return self._items.pop(0) if BUG == "pop" else self._items.pop()

    def peek(self):
        if not self._items:
            raise IndexError("peek at empty stack")
        return self._items[0] if BUG == "peek" else self._items[-1]

    def is_empty(self):
        return not self._items

    def __len__(self):
        return len(self._items)


class TestStack(unittest.TestCase):
    def setUp(self):
        self.stack = Stack()

    def test_push_then_pop(self):
        self.stack.push(1)
        self.stack.push(2)
        self.assertEqual(self.stack.pop(), 2)
        self.assertEqual(self.stack.pop(), 1)

    def test_peek_does_not_remove(self):
        self.stack.push(1)
        self.stack.push(2)
        self.assertEqual(self.stack.peek(), 2)
        self.assertEqual(len(self.stack), 2)

    def test_pop_empty_raises(self):
```

```
        with self.assertRaises(IndexError):
            self.stack.pop()

    def test_len_and_is_empty(self):
        self.assertTrue(self.stack.is_empty())
        self.assertEqual(len(self.stack), 0)
        self.stack.push(5)
        self.assertEqual(len(self.stack), 1)
        self.assertFalse(self.stack.is_empty())


suite = unittest.defaultTestLoader.loadTestsFromTestCase(TestStack)
result = unittest.TextTestRunner(stream=io.StringIO(), verbosity=0).run(suite)
print("ran", result.testsRun)
print("failures", len(result.failures))
print("errors", len(result.errors))
print("ok", result.wasSuccessful())
for name in sorted(test.id().rsplit(".", 1)[-1] for test, _ in result.failures):
    print("failed", name)
```

EXERCISE 2

A subTest table

Write `checked_div(a, b)` returning `a / b` and raising `ZeroDivisionError` when `b` is 0. Read rows `a b expected` until EOF, where `expected` is a number or the word `raises`. Write a `TestCase` with `test_table`, which loops over the numeric rows inside `with self.subTest(a=a, b=b):` asserting `checked_div(a, b) == float(expected)`, and `test_raises`, which loops over the `raises` rows inside `subTest` asserting `assertRaises(ZeroDivisionError)`. Run the suite on a `StringIO` stream and print `ran <n>`, `failures <f>`, `errors <e>`, `ok <True/False>`.

Input: one row per line. **Output:** four lines.

```
10 2 5
7 2 3.5
1 0 raises
9 3 4
```

prints

```
ran 2
failures 1
errors 0
ok False
```

INPUT 1

```
10 2 5
7 2 3.5
1 0 raises
9 3 4
```

OUTPUT 1

```
ran 2
failures 1
errors 0
ok False
```

REFERENCE SOLUTION

```
import io
import sys
import unittest

ROWS = [line.split() for line in sys.stdin if len(line.split()) == 3]
NUMERIC = [(int(a), int(b), float(e)) for a, b, e in ROWS if e != "raises"]
RAISING = [(int(a), int(b)) for a, b, e in ROWS if e == "raises"]


def checked_div(a, b):
    if b == 0:
        raise ZeroDivisionError("b must not be zero")
    return a / b


class TestDiv(unittest.TestCase):
    def test_table(self):
        for a, b, expected in NUMERIC:
            with self.subTest(a=a, b=b):
                self.assertEqual(checked_div(a, b), expected)

    def test_raises(self):
        for a, b in RAISING:
            with self.subTest(a=a, b=b):
                with self.assertRaises(ZeroDivisionError):
                    checked_div(a, b)


suite = unittest.defaultTestLoader.loadTestsFromTestCase(TestDiv)
result = unittest.TextTestRunner(stream=io.StringIO(), verbosity=0).run(suite)
print("ran", result.testsRun)
print("failures", len(result.failures))
print("errors", len(result.errors))
print("ok", result.wasSuccessful())
```

QUIZ

1. Which methods does `unittest` run as tests?

A. Every method of a `TestCase`

B. Methods whose names start with `test`

C. Methods decorated with `@test`

D. Methods named in `__tests__`

2. Why prefer `assertEqual(a, b)` over `assertTrue(a == b)`?

A. It is faster

B. Its failure message shows both values and a diff

C. It handles floats

D. It is the only one that works

3. When does `setUp` run?

A. Once per class

B. Before every test method

C. Once per module

D. Only if called

4. What is the effect of `subTest` on a failing table?

A. The loop stops at the first failure

B. Every row is checked and each failure is reported with its parameters

C. Failures are ignored

D. Each row counts in `testsRun`

5. Why run a suite with `TextTestRunner(stream=io.StringIO())` in a program?

A. It is faster

B. To keep the runner's report (with its timing) off stdout and read the `TestResult` instead

C. To skip failing tests

D. It is required

ANSWER KEY · UNITTEST — TESTCASE, ASSERTIONS, FIXTURES, SUBTEST AND RUNNING SUITES

1. B A method named `check_total` is silently never run.

2. B `assertTrue` can only say "False is not true".

3. B Every test starts from fresh state; `setUpClass` is the once-per-class hook.

4. B The method is one test in `testsRun`; each failing subtest is its own entry in `failures`.

5. B `unittest.main()` also calls `sys.exit`, which a program that continues does not want.

LESSON 3 · 13 MIN

pytest in outline — plain asserts, fixtures, parametrize and the command line

pytest is the test runner most Python projects use: tests are plain functions named `test_*`, assertions are the `assert` statement with the failing expression's values shown on failure, fixtures are functions whose return value is injected by parameter name, and `@pytest.mark.parametrize` turns one test into a table. It is a third-party package (`pip install pytest`) and the judge does not have it, so this lesson is reading and the exercises rebuild its two central mechanisms — parametrisation and fixtures with teardown — in plain Python, which is the best way to understand what pytest does for you. It covers the assert rewriting, fixtures and their scopes, `parametrize`, `raises`, markers and the command-line flags you use daily, and how pytest runs `unittest` suites unchanged.

A test file

```python
# test_slugify.py
from slugify import slugify

def test_lowercases_and_joins():
    assert slugify("Hello World") == "hello-world"

def test_collapses_whitespace():
    assert slugify("  a   b ") == "a-b"
```

`pytest` in the project directory finds `test_*.py` and `*_test.py` files, collects `test_*` functions and `Test*` classes (no base class needed), runs them, and prints a dot per pass. On failure it shows the assertion with every sub-expression's value:

```
    def test_collapses_whitespace():
>       assert slugify("  a   b ") == "a-b"
E       AssertionError: assert 'a--b' == 'a-b'
E         - a-b
E         + a--b
```

That is *assertion rewriting*: pytest recompiles test modules so a plain `assert` reports like a specialised method. There is no assertion family to learn.

Fixtures

```python
import pytest

@pytest.fixture
def inventory():
    inv = Inventory()
    inv.add("bolt", 10)
    yield inv                       # the test runs here
    inv.close()                     # teardown, even if the test failed

def test_remove(inventory):         # injected by parameter name
    inventory.remove("bolt", 4)
    assert inventory.count("bolt") == 6
```

A fixture is a function decorated with `@pytest.fixture`; a test that names it as a parameter receives its value. A fixture that `yield`s runs its remainder as teardown. Fixtures can use other fixtures, and `scope="module"` or `"session"` makes one instance serve many tests (a database connection). Fixtures in `conftest.py` are available to every test in that directory tree without imports. Built-in fixtures: `tmp_path` (a fresh temporary directory as a `Path`), `monkeypatch` (`monkeypatch.setattr(obj, "name", value)`, `setenv`, all undone afterwards), `capsys` (captured stdout/stderr via `capsys.readouterr()`), `caplog` (captured log records).

parametrize

```python
@pytest.mark.parametrize("text, expected", [
    ("1h", 3600),
    ("30m", 1800),
    ("1h30m", 5400),
    pytest.param("", 0, id="empty"),
])
def test_parse_duration(text, expected):
    assert parse_duration(text) == expected
```

One function, four tests, each reported separately as `test_parse_duration[1h-3600]`, and each failing independently. Stacking two `parametrize` decorators produces the cross product. This is `subTest` without the indentation and with proper per-case identity.

raises, approx, markers

```
with pytest.raises(ValueError, match="name"):
    User(name="")

assert total == pytest.approx(99.0)                  # float tolerance

@pytest.mark.skipif(sys.platform == "win32", reason="posix only")
@pytest.mark.xfail(reason="bug #42")
@pytest.mark.slow                                    # custom marker; select w
```

`pytest.raises` is the `assertRaises` twin, `match` is a regex on the message; `approx` compares floats and sequences of floats with relative tolerance; markers tag tests for selection, skipping and expected failure.

The command line

Flag	Effect
`pytest -x`	stop at the first failure
`pytest -k "slug and not empty"`	select by name expression
`pytest path/test_x.py::test_name`	one test
`pytest --lf` / `--ff`	rerun last failures / failures first
`pytest -v` / `-q`	verbose / quiet
`pytest -s`	do not capture stdout (see prints)
`pytest --pdb`	drop into the debugger on failure
`pytest -m slow` / `-m "not slow"`	by marker
`pytest --durations=10`	the slowest tests

Configuration lives in `pyproject.toml` under `[tool.pytest.ini_options]` (`testpaths`, `addopts`, registered `markers`). Plugins add flags: `pytest-cov` for coverage, `pytest-xdist` for `-n auto` parallel runs, `pytest-asyncio` for `async def` tests.

unittest under pytest

pytest runs `unittest.TestCase` classes as they are — `setUp`, assertion methods and all — so a code base can migrate one file at a time, and a new test file beside old ones can use the plain style. The two styles differ in fixtures (methods versus injected functions) and assertions (methods versus `assert`); the discipline of the previous lesson — one behaviour per test, edges and errors, fresh state — is identical.

Rebuilding the mechanisms

Both of pytest's central ideas are small. Parametrisation is a decorator that attaches a list of argument tuples to a function, and a runner that calls the function once per tuple, catching `AssertionError` and reporting each case by name. A yield-fixture is a generator: the runner calls `next()` to get the value, runs the test, then calls `next()` again inside a `finally` so teardown runs whether the test passed or not. The exercises build exactly these, which is also what you would write to run a table of checks in a script with no framework available.

Pitfalls

- A fixture that does setup with `return` and expects teardown to run — only `yield` fixtures tear down.
- `assert` in production code for validation — `python -O` strips it; use exceptions.
- A parametrize id that is unreadable in the report — `pytest.param(..., id="…")`.
- Tests that pass only because of a session-scoped fixture's leftover state.
- Forgetting `-s` and wondering where the prints went.
- Mixing `unittest` assertion methods into plain functions (`self` does not exist there).

Key takeaways

- pytest collects `test_*` functions and reports plain `assert` with values; no base class, no assertion family.
- Fixtures are injected by parameter name; `yield` fixtures tear down; `tmp_path`, `monkeypatch`, `capsys` are built in; `conftest.py` shares them.

`@pytest.mark.parametrize` makes a table into independent tests; `raises`, `approx` and markers cover exceptions, floats and selection.

`-x`, `-k`, `--lf`, `-s`, `--pdb` are the daily flags; configuration lives in `pyproject.toml`.

Parametrisation is "call once per tuple"; a yield fixture is a generator driven by `next()` around the test — small enough to rebuild when pytest is not there.

EXERCISE 1

parametrize in plain Python

Read rows `a b expected` until EOF. Write a decorator `cases(*rows)` that stores the rows on the function as `fn.cases`, and a runner `run(*tests)` that, for each test function and each of its rows in order, calls `fn(*row)`; on success it prints `<name>[<i>] PASS`, on `AssertionError` it prints `<name>[<i>] FAIL <message>`. Define `add(a, b)`, then `test_add(a, b, expected)` asserting `add(a, b) == expected` with the message `got <result>`, and `test_commutes(a, b, expected)` asserting `add(a, b) == add(b, a)`; decorate both with the rows and run them. Finish with `<p> passed, <f> failed`.

Input: one row per line. **Output:** one line per test and row, then the summary.

```
1 2 3
2 2 5
0 0 0
```

prints

```
test_add[0] PASS
test_add[1] FAIL got 4
test_add[2] PASS
test_commutes[0] PASS
test_commutes[1] PASS
test_commutes[2] PASS
5 passed, 1 failed
```

INPUT 1

```
1 2 3
2 2 5
0 0 0
```

OUTPUT 1

```
test_add[0] PASS
test_add[1] FAIL got 4
test_add[2] PASS
test_commutes[0] PASS
test_commutes[1] PASS
test_commutes[2] PASS
5 passed, 1 failed
```

REFERENCE SOLUTION

```python
import sys

ROWS = [tuple(int(x) for x in line.split()) for line in sys.stdin if len(line.split(

def cases(*rows):
    def decorator(fn):
        fn.cases = list(rows)
        return fn
    return decorator


def run(*tests):
    passed = failed = 0
    for fn in tests:
        for i, row in enumerate(fn.cases):
            try:
                fn(*row)
            except AssertionError as e:
                failed += 1
                print(f"{fn.__name__}[{i}] FAIL {e}")
            else:
                passed += 1
                print(f"{fn.__name__}[{i}] PASS")
    return passed, failed


def add(a, b):
    return a + b


@cases(*ROWS)
def test_add(a, b, expected):
    result = add(a, b)
    assert result == expected, f"got {result}"


@cases(*ROWS)
def test_commutes(a, b, expected):
    assert add(a, b) == add(b, a)


passed, failed = run(test_add, test_commutes)
print(f"{passed} passed, {failed} failed")
```

EXERCISE 2

A yield fixture

Write a generator fixture `connection()` that prints `setup`, yields the dict `{"open": True}`, then sets `"open"` to `False` and prints `teardown`. Write `run_with(fixture, test)` that drives the generator with `next`, calls `test(value)`, prints `<test name> PASS` or `<test name> FAIL <message>` on `AssertionError`, and always runs the teardown in a `finally`. Read lines `ok` or `fail` until EOF and build one test per line named `test_<i>` (use a factory so each has its own name via `__name__`) that asserts `conn["open"]` and, for `fail`, then `assert False, "forced failure"`. Run them in order and finish with `<p> passed, <f> failed`.

Input: one marker per line. **Output:** setup/result/teardown per test, then the summary.

```
ok
fail
```

prints

```
setup
test_0 PASS
teardown
setup
test_1 FAIL forced failure
teardown
1 passed, 1 failed
```

INPUT 1

```
ok
fail
```

OUTPUT 1

```
setup
test_0 PASS
teardown
setup
test_1 FAIL forced failure
teardown
1 passed, 1 failed
```

REFERENCE SOLUTION

```
import sys


def connection():
    print("setup")
    conn = {"open": True}
    yield conn
    conn["open"] = False
    print("teardown")


def run_with(fixture, test):
    """Return True when the test passed."""
    gen = fixture()
    value = next(gen)
    try:
        test(value)
    except AssertionError as e:
        print(f"{test.__name__} FAIL {e}")
        return False
    else:
        print(f"{test.__name__} PASS")
        return True
    finally:
        next(gen, None)


def make_test(i, should_fail):
    def test(conn):
        assert conn["open"]
        if should_fail:
            assert False, "forced failure"
    test.__name__ = f"test_{i}"
    return test


markers = [line.strip() for line in sys.stdin if line.strip()]
tests = [make_test(i, marker == "fail") for i, marker in enumerate(markers)]
results = [run_with(connection, test) for test in tests]
print(f"{sum(results)} passed, {len(results) - sum(results)} failed")
```

QUIZ

1. How does pytest show useful failure messages from a plain `assert`?

A. It cannot

B. It rewrites the assertion at import time to report each sub-expression's value

C. It requires `assertEqual`

D. It uses `repr` of the function

2. How does a test receive a fixture?

A. By importing it

B. By naming it as a parameter

C. By calling it

D. Through `self`

3. Which fixture form runs teardown?

A. One that `return`s

B. One that `yield`s — the code after the `yield` runs after the test

C. Any fixture

D. One with `scope="session"`

4. What does `@pytest.mark.parametrize("a, b", [(1, 2), (3, 4)])` produce?

A. One test run twice inside a loop

B. Two independent tests, each reported and failing on its own

C. A skipped test

D. A fixture

5. Which flag reruns only the tests that failed last time?

A. `-x`

B. `--lf`

C. `-k`

D. `-s`

ANSWER KEY · PYTEST IN OUTLINE — PLAIN ASSERTS, FIXTURES, PARAMETRIZE AND THE COMMAND LINE

1. B Assertion rewriting is why pytest needs no assertion family.

2. B Injection by name; fixtures can depend on other fixtures the same way.

3. B The runner resumes the generator after the test, whether it passed or failed.

4. B That is `subTest` with proper per-case identity.

5. B `-x` stops at the first failure; `-k` selects by name; `-s` shows prints.

LESSON 4 · 15 MIN

Mocking and test doubles — Mock, patch, side_effect and where to patch

A unit test wants to run one piece of code without the network, the database, the clock or the file system it depends on. A *test double* stands in for the dependency: it answers with canned values, records what was asked of it, or both. `unittest.mock` provides `Mock` and `MagicMock` objects that accept any call and remember it, `patch` to swap a name for a mock during a test, `side_effect` to script answers and exceptions, and `spec` / `autospec` to keep a mock honest about the real interface. This lesson covers the vocabulary of doubles, the `Mock` API, `patch` in its three forms and the rule about *where* to patch, scripted behaviour, and the judgement of when a fake beats a mock.

The vocabulary

Double	What it does	Example
Stub	returns canned answers	a `get_rate()` that returns `0.1`
Fake	a working lightweight implementation	an in-memory dict standing in for a key-value store
Spy	records calls, then delegates or returns	a sender that appends to a list
Mock	records calls and can assert on them	`Mock()` with `assert_called_once_with`

Injected dependencies (lesson 1) make any of them pluggable: a class that takes `sender` in `__init__` can be given a `Mock()` in a test and the real `SmtpSender` in production.

Mock

```python
from unittest.mock import Mock

sender = Mock()
sender.send("ada@example.com", "hi")     # any attribute, any call: accepted
sender.send.assert_called_once_with("ada@example.com", "hi")
sender.send.call_count                   # 1
sender.send.call_args                    # call('ada@example.com', 'hi')
sender.send.call_args_list               # [call('ada@example.com', 'hi')]
sender.send.call_args.args               # ('ada@example.com', 'hi')

rate = Mock(return_value=0.1)
rate()                                   # 0.1
rate.assert_called()
```

A `Mock` creates attributes on first access, each itself a `Mock`; every call returns `return_value` (another `Mock` by default) and is recorded. The assertion methods — `assert_called`, `assert_called_once`, `assert_called_with`, `assert_called_once_with`, `assert_any_call`, `assert_not_called` — raise `AssertionError` with a message showing the actual calls. `MagicMock` additionally implements the dunder methods (`__len__`, `__iter__`, `__enter__`), so it can stand in for a context manager or a container. `reset_mock()` clears the record.

---

side_effect: scripted behaviour

```python
fetch = Mock(side_effect=[TimeoutError(), TimeoutError(), {"price": 42}])
fetch()          # raises TimeoutError
fetch()          # raises TimeoutError
fetch()          # {'price': 42}

parse = Mock(side_effect=lambda s: int(s) * 2)      # a function computes the
parse("21")      # 42

boom = Mock(side_effect=ValueError("bad"))          # always raises
```

`side_effect` as an exception raises it; as a list, each call returns the next item (an exception instance in the list is raised); as a callable, it is called with the arguments. That is how a retry loop is tested: two failures then a success, and `call_count == 3` afterwards.

---

patch

```python
from unittest.mock import patch

# 1. decorator: the mock is passed as the last positional argument
@patch("billing.fetch_rate", return_value=0.2)
def test_total(mock_rate):
    assert total(100) == 120
    mock_rate.assert_called_once()

# 2. context manager
def test_total():
    with patch("billing.fetch_rate", return_value=0.2) as mock_rate:
        assert total(100) == 120

# 3. patch.object: an attribute on an object you hold
with patch.object(Clock, "now", return_value=datetime(2024, 1, 1)):
    ...
```

`patch(target)` replaces the object at the dotted path with a `MagicMock` (or `new=` something) for the duration and restores it afterwards, exception or not. `patch.dict(os.environ, {"MODE": "test"})` patches a mapping. Several decorators stack; their mocks arrive bottom-up.

Where to patch

The target is *where the name is looked up*, not where it is defined. If `billing.py` does `from rates import fetch_rate`, then `billing.fetch_rate` is the name its code reads, and patching `rates.fetch_rate` changes nothing for it. If it does `import rates` and calls `rates.fetch_rate()`, patch `rates.fetch_rate`. The rule: patch the name in the module under test. Most "my patch has no effect" bugs are this.

---

spec and autospec

```python
sender = Mock(spec=SmtpSender)          # only SmtpSender's attributes exist
sender.snd("x")                         # AttributeError — a typo is caught

with patch("billing.fetch_rate", autospec=True) as mock_rate:
    mock_rate("usd", "eur", "extra")    # TypeError — signature enforced
```

A bare `Mock` accepts anything, so a test can pass against a method that was renamed or a call with the wrong arguments. `spec=Class` restricts attributes; `autospec=True` on `patch` also checks call signatures. Use them by default for anything with a real interface.

Fakes over mocks

A mock checks *interaction*: "send was called once with these arguments". A fake checks *outcome*: an in-memory store that the code writes to, and the test reads back. When the collaborator has state and behaviour (a repository, a queue, a cache), a small fake — a class with the same methods over a dict — produces tests that survive refactors and read like the real thing; a mock of every method produces tests that assert the implementation. Mocks are right for edges of the system — the network, the clock, email — where the interaction is the behaviour.

Pitfalls

- Patching where the function is defined instead of where it is used.
- A test that passes because the mock returns a `Mock` that is truthy and comparable to nothing — assert on the value used.
- No `spec`, so a renamed method is never noticed.
- Mocking the class under test.
- Forgetting that `patch` as a decorator passes the mock as an argument.
- Asserting `called_with` when the code should have been tested by its result.

Key takeaways

- Stubs answer, fakes work, spies record, mocks record and assert; injection makes them pluggable.
- `Mock` accepts any call, records `call_args` / `call_count`, and asserts with `assert_called_once_with` and friends; `MagicMock` adds dunders.
- `side_effect` scripts a sequence, a function or an exception; `return_value` is the constant answer.
- `patch` / `patch.object` / `patch.dict` swap a name for the test's duration — at the place the code under test looks it up.

`spec` / `autospec` keep mocks honest; prefer a fake when the collaborator has state and the outcome is what matters.

EXERCISE 1

Spy on the notifier

Write `OrderService` taking a `sender` in `__init__`; its `place(order_id, amount)` calls `sender.notify(order_id, amount)` when `amount` is at least 100 and returns `amount`. Create the service with a `unittest.mock.Mock()` as the sender, read lines `<id> <amount>` (integers) until EOF and place each order. Then print `notified <sender.notify.call_count>`, one line `call <id> <amount>` per entry of `call_args_list` (use `.args`), and `total <sum of the amounts returned>`.

Input: one order per line. **Output:** the count, the recorded calls, the total.

```
1 50
2 150
3 100
```

prints

```
notified 2
call 2 150
call 3 100
total 300
```

INPUT 1

```
1 50
2 150
3 100
```

OUTPUT 1

```
notified 2
call 2 150
call 3 100
total 300
```

REFERENCE SOLUTION

```
import sys
from unittest.mock import Mock


class OrderService:
    def __init__(self, sender):
        self.sender = sender

    def place(self, order_id, amount):
        if amount ⩾ 100:
            self.sender.notify(order_id, amount)
        return amount


sender = Mock()
service = OrderService(sender)
total = 0
for line in sys.stdin:
    parts = line.split()
    if len(parts) ≠ 2:
        continue
    total += service.place(int(parts[0]), int(parts[1]))

print("notified", sender.notify.call_count)
for recorded in sender.notify.call_args_list:
    order_id, amount = recorded.args
    print(f"call {order_id} {amount}")
print("total", total)
```

EXERCISE 2

Script the network with side_effect

The starter's `fetch_price(symbol)` raises `RuntimeError("network")` — the real thing must never run in a test. Write `average_price(symbol, n)` that calls `fetch_price(symbol)` `n` times and returns the mean. Read a symbol on the first line and prices on the second; inside `with patch("__main__.fetch_price", side_effect=prices) as mock:` compute the average of `len(prices)` fetches and print `average <x.2f>`, `calls <mock.call_count>` and `called with <mock.call_args.args[0]>`. After the block, call the real `fetch_price` inside `try` and print `real restored` when it raises `RuntimeError`.

Input: the symbol, then the prices. **Output:** four lines.

```
acme
10 20 30
```

prints

```
average 20.00
calls 3
called with acme
real restored
```

INPUT 1

```
acme
10 20 30
```

OUTPUT 1

```
average 20.00
calls 3
called with acme
real restored
```

REFERENCE SOLUTION

```
from unittest.mock import patch


def fetch_price(symbol):
    raise RuntimeError("network")


def average_price(symbol, n):
    total = sum(fetch_price(symbol) for _ in range(n))
    return total / n


symbol = input().strip()
prices = [float(x) for x in input().split()]

with patch("__main__.fetch_price", side_effect=prices) as mock:
    average = average_price(symbol, len(prices))
    print(f"average {average:.2f}")
    print("calls", mock.call_count)
    print("called with", mock.call_args.args[0])

try:
    fetch_price(symbol)
except RuntimeError:
    print("real restored")
```

QUIZ

1. What does `Mock().anything(1, 2)` do?

A. Raises `AttributeError`

B. Creates the attribute, accepts the call, records it, and returns another `Mock`

C. Returns `None`

D. Calls the real method

2. What does `Mock(side_effect=[ValueError(), 3])` do across two calls?

A. Returns the list

B. Raises `ValueError` on the first call, returns 3 on the second

C. Returns 3 twice

D. Raises twice

3. `billing.py` does `from rates import fetch_rate`. Which target patches what `billing` calls?

A. `rates.fetch_rate`

B. `billing.fetch_rate`

C. Either

D. `__main__.fetch_rate`

4. What does `autospec=True` add to `patch`?

A. Speed

B. The mock has only the real attributes and enforces the real call signatures

C. Automatic return values

D. Nothing

5. When is a fake better than a mock?

A. Never

B. When the collaborator has state and the outcome, not the interaction, is what the test should check

C. For the network

D. For the clock

ANSWER KEY · MOCKING AND TEST DOUBLES — MOCK, PATCH, SIDE_EFFECT AND WHERE TO PATCH

1. B A bare `Mock` accepts everything — which is why `spec` exists.

2. B A list is played back one item per call; exception instances are raised.

3. B Patch where the name is looked up — the module under test — not where it is defined.

4. B A typo or a wrong argument count fails instead of being accepted silently.

5. B An in-memory repository survives refactors; a mock of every method asserts the implementation.

LESSON 5 · 15 MIN

Debugging — reading tracebacks, logging, pdb and the method

Debugging is finding out why the program did something other than what you believed it would, and most of the skill is in reading what Python already tells you: a traceback names the exact line and the chain of calls that reached it, an exception chain records what was being handled when the second error occurred, and a logger records the path the program took without stopping it. When that is not enough, `pdb` stops the program at a line and lets you inspect it. This lesson covers reading a traceback bottom-up, chained exceptions, `traceback` and `logging` as diagnostic tools, `breakpoint()` and the debugger commands, `assert` and the `-X dev` and `-W error` switches, and the method — reproduce, minimise, hypothesise, bisect — that turns a mystery into a fix.

Reading a traceback

```
Traceback (most recent call last):
  File "app.py", line 21, in <module>
    main()
  File "app.py", line 17, in main
    report(load("data.csv"))
  File "app.py", line 9, in load
    return [parse(line) for line in f]
  File "app.py", line 5, in parse
    return int(line.split(",")[1])
ValueError: invalid literal for int() with base 10: 'n/a'
```

Read from the bottom: the last line is the exception type and message — the *what*. The frame above it is *where* it was raised (`parse` , line 5). The frames above that are *how* execution got there, outermost first. Each frame shows file, line number, function and the source line. Your own code's frames are the ones to focus on; frames inside site-packages usually mean your code passed something wrong to a library, and the frame of yours nearest the bottom is the suspect. The message is the second clue: `'n/a'` says which input broke the assumption.

Chained exceptions

```
Traceback (most recent call last):
  File "app.py", line 12, in fetch
    return cache[key]
KeyError: 'user:7'

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  ...
  File "app.py", line 14, in fetch
    return db.get(key)
ConnectionError: database unavailable
```

"During handling of the above exception" means the second error happened inside an `except` block; "The above exception was the direct cause" means `raise X from e`. Both tracebacks matter: the first explains why the handler ran, the second why it failed. `raise ... from None` suppresses the chain when the first error is an implementation detail.

---

traceback and logging

```python
import logging, traceback

log = logging.getLogger("app")

try:
    run()
except Exception:
    log.exception("run failed")            # message + full traceback at ERROR
    text = traceback.format_exc()          # the traceback as a string, for a
```

`logging` is the diagnostic channel of a program that keeps running: `basicConfig(level=logging.DEBUG, format="%(levelname)s %(name)s: %(message)s")` sets up a handler to stderr; `log.debug/info/warning/error/critical` emit at levels; the threshold decides what is shown, so debug detail can stay in the code and be switched on in the field. Loggers are hierarchical by dotted name; a handler can be a file, a stream or your own `Handler` subclass with an `emit(record)` method. `print` is for the program's output; `logging` is for its diary. In a judged program a `StreamHandler(sys.stdout)` with a format that has no timestamp is deterministic; `%(asctime)s` is not.

pdb

```
def parse(line):
    breakpoint()                          # stops here; PYTHONBREAKPOINT=0 c
    return int(line.split(",")[1])
```

`breakpoint()` opens `pdb` at that line. The commands: `n` (next line), `s` (step into a call), `c` (continue), `r` (return from the function), `l` (list source), `ll` (whole function), `p expr` / `pp expr` (print), `w` (where — the stack), `u` / `d` (up/down a frame), `b file:line` (set a breakpoint), `q` (quit). `python -m pdb app.py` starts under the debugger; `python -c "import pdb, app; pdb.pm()"` after a crash, or `pdb.post_mortem()` in an `except`, inspects the frames of the exception that just happened — the fastest way to see local variables at the point of failure.

assert, -X dev, -W error

`assert cond, "message"` documents an invariant and raises `AssertionError` when it is false; `python -O` removes every assert, so it is for checking *your own* logic, never for validating input. `python -X dev` turns on development mode: warnings shown, `ResourceWarning` for unclosed files, extra checks in `asyncio`. `python -W error` turns warnings into exceptions so a deprecation cannot be ignored. `python -X faulthandler` (or `faulthandler.enable()`) prints a traceback on a hard crash such as a segfault in a C extension.

The method

Reproduce it: a command or a test that fails every time. An intermittent failure is a different bug (usually a race or an unseeded random).

Minimise: the smallest input and the fewest lines that still fail. Halving the input repeatedly finds the offending record; deleting code that is not involved shrinks the search.

Hypothesise: from the traceback and the message, one specific claim ("`parse` assumes column 1 is numeric"). Check it with a print of `f"{line=}"` or a breakpoint — one variable, not twenty.

Bisect across history when the failure is new: `git bisect` between the last good and first bad commit finds the change in $\log_2$ steps.

Fix, then add the regression test from step 1, then remove the prints.

`print(f"{x=}")` (the `=` specifier prints the expression and its `repr`) is the humble tool that solves most bugs; the discipline is to print `repr`, not `str`, so `'1 '` with a trailing space is visible.

Pitfalls

Reading only the last line of a traceback and guessing where.

Ignoring the first traceback of a chain.

`except Exception: pass` hiding the error that would have explained everything.

Validation with `assert` (disappears under `-O`).

Debug prints of `str(x)` that hide whitespace and types.

Changing three things at once and not knowing which one fixed it.

Key takeaways

Read a traceback bottom-up: type and message, then the raising frame, then the path; your own nearest frame is the suspect.

Chained tracebacks show the error that was being handled and the one that followed; `from` and `from None` control the chain.

`logging` is the running program's diary with levels and handlers; `log.exception` and `traceback.format_exc` capture tracebacks.

`breakpoint()` and `pdb.pm()` inspect live frames; `n s c p w u d` are the commands.

Reproduce, minimise, hypothesise, bisect, fix, regression-test; print `repr`, never validate with `assert`.

EXERCISE 1

A logging pipeline

Read a threshold level name on the first line (`DEBUG`, `INFO`, `WARNING`, `ERROR` or `CRITICAL`), then lines `<LEVEL> <message>` until EOF. Configure a logger named `app` (with `propagate = False`) that has a `StreamHandler` on `sys.stdout` using the format `%(levelname)s %(name)s: %(message)s`, set to the threshold; attach a `logging.Filter` subclass that counts the records that reach the

handler. Emit each line with `logger.log(level, message)`, then print `emitted <count> of <total>`.

Input: the threshold, then records. **Output:** the records at or above the threshold, then the count.

```
WARNING
INFO starting
WARNING disk at 91%
ERROR write failed
DEBUG payload {}
```

prints

```
WARNING app: disk at 91%
ERROR app: write failed
emitted 2 of 4
```

INPUT 1

```
WARNING
INFO starting
WARNING disk at 91%
ERROR write failed
DEBUG payload {}
```

OUTPUT 1

```
WARNING app: disk at 91%
ERROR app: write failed
emitted 2 of 4
```

REFERENCE SOLUTION

```
import logging
import sys


class Counting(logging.Filter):
    def __init__(self):
        super().__init__()
        self.count = 0

    def filter(self, record):
        self.count += 1
        return True


threshold = sys.stdin.readline().strip()
logger = logging.getLogger("app")
logger.propagate = False
logger.setLevel(getattr(logging, threshold))
handler = logging.StreamHandler(sys.stdout)
handler.setFormatter(logging.Formatter("%(levelname)s %(name)s: %(message)s"))
counting = Counting()
handler.addFilter(counting)
logger.addHandler(handler)

total = 0
for line in sys.stdin:
    level, _, message = line.rstrip().partition(" ")
    if not level:
        continue
    total += 1
    logger.log(getattr(logging, level), message)

print(f"emitted {counting.count} of {total}")
```

EXERCISE 2

Traceback reader

Read a Python traceback from standard input (possibly a chained one with two sections). Consider only the last section — the text after the final `Traceback (most recent call last):`. Extract its frames from lines of the form `File "<file>", line <n>, in <func>` and the final exception line. Print `error <type>: <message>` (or `error <type>` when there is no message), `raised at <file>: <line> in <func>` for the last frame, `frames <count>`, and `chained <True/False>` — true when the text contains `During handling of the above exception` or `The above exception was the direct cause`.

Input: a traceback. **Output:** four lines.

```
Traceback (most recent call last):
  File "app.py", line 21, in <module>
    main()
  File "app.py", line 17, in main
    report(load("data.csv"))
  File "app.py", line 9, in load
    return [parse(line) for line in f]
  File "app.py", line 5, in parse
    return int(line.split(",")[1])
ValueError: invalid literal for int() with base 10: 'n/a'
```

prints

```
error ValueError: invalid literal for int() with base 10: 'n/a'
raised at app.py:5 in parse
frames 4
chained False
```

INPUT 1

```
Traceback (most recent call last):
  File "app.py", line 21, in <module>
    main()
  File "app.py", line 17, in main
    report(load("data.csv"))
  File "app.py", line 9, in load
    return [parse(line) for line in f]
  File "app.py", line 5, in parse
    return int(line.split(",")[1])
ValueError: invalid literal for int()
with base 10: 'n/a'
```

OUTPUT 1

```
error ValueError: invalid literal for
int() with base 10: 'n/a'
raised at app.py:5 in parse
frames 4
chained False
```

REFERENCE SOLUTION

```
import re
import sys

FRAME = re.compile(r'^\s*File "(.+?)", line (\d+), in (.+)$')
HEADER = "Traceback (most recent call last):"
CHAIN_MARKERS = ("During handling of the above exception", "The above exception was

text = sys.stdin.read()
last_section = text.rsplit(HEADER, 1)[-1]
lines = [line.rstrip() for line in last_section.splitlines() if line.strip()]

frames = [m.groups() for m in map(FRAME.match, lines) if m]
exc_line = lines[-1].strip()
exc_type, sep, message = exc_line.partition(": ")
chained = any(marker in text for marker in CHAIN_MARKERS)

print(f"error {exc_type}: {message}" if sep else f"error {exc_type}")
file, line_no, func = frames[-1]
print(f"raised at {file}:{line_no} in {func}")
print("frames", len(frames))
print("chained", chained)
```

QUIZ

1. In a traceback, where is the line that raised the exception?

A. The first frame listed

B. The last frame, just above the exception line

C. The `<module>` frame

D. It is not shown

2. What does "During handling of the above exception, another exception occurred" mean?

A. The exception was re-raised

B. A second exception was raised inside the `except` block handling the first

C. The exception was suppressed

D. Two threads crashed

3. Why not validate user input with `assert`?

A. It is slow

B. `python -O` removes every `assert`, so the check disappears

C. It raises the wrong type

D. It cannot take a message

4. Which `pdb` command steps *into* a function call?

A. `n`

B. `s`

C. `c`

D. `w`

5. What is the first step of the debugging method?

A. Add prints everywhere

B. Reproduce the failure reliably

C. Change the suspected line

D. Bisect

ANSWER KEY · DEBUGGING — READING TRACEBACKS, LOGGING, PDB AND THE METHOD

1. B Frames are listed outermost first; read from the bottom.

2. B "The above exception was the direct cause" is the `raise … from e` form.

3. B Asserts document your own invariants; input validation raises `ValueError`.

4. B `n` steps over, `c` continues, `w` shows the stack.

5. B Then minimise, hypothesise, bisect if the failure is new, fix, and add the regression test.

CHECKPOINT · 25 MIN

Checkpoint — Testing and debugging

This checkpoint covers the whole module: the testing mindset — kinds of test, arrange–act–assert, what to test and designing for testability by injecting the clock, seed and collaborators; `unittest` with its assertion family, fixtures, `subTest` and in-process runs; pytest's plain asserts, fixtures and `parametrize` and the mechanisms behind them; test doubles with `Mock`, `side_effect`, `patch` and the rule about where to patch; and debugging by reading tracebacks, logging, `pdb` and the reproduce–minimise–bisect method.

How it works. Twelve questions and three programs. You need 70% on the questions and every program accepted to clear the module. You can retake it as often as you like; your best score counts.

Before you start, make sure you can answer these from memory:

- What are the three parts of a test, and why one behaviour per test?
- Why is `assertEqual(a, b)` better than `assertTrue(a == b)`?
- What does `subTest` change about a failing table?
- How does a `yield` fixture guarantee teardown?
- What does `side_effect=[TimeoutError(), 42]` make a mock do?
- Where must `patch` target a name for the code under test to see the mock?
- In which order do you read a traceback, and what does "During handling of the above exception" mean?

The three programs are a `unittest` suite for a bank account that must catch each of three injected bugs by name, a retry helper tested against a scripted `Mock` with a counted number of attempts, and a counting `logging.Handler` that classifies a stream of records by level and reports what a threshold would have shown.

EXERCISE 1

A suite against injected bugs

The starter defines `BankAccount` with a bug injected by the first input line: `none`, `negative` (deposit accepts non-positive amounts), `overdraw` (withdraw allows going below zero) or `interest` (`add_interest` adds the rate instead of the percentage). Write `TestBankAccount` with exactly: `test_deposit_increases_balance` (100 + deposit 50 → 150), `test_deposit_rejects_non_positive`

(`assertRaises(ValueError)` for `deposit(0)` and for `deposit(-5)`), `test_withdraw_rejects_overdraw` (balance 100, `assertRaises(ValueError)` for `withdraw(150)`) and `test_interest` (balance 200, `add_interest(0.05)` , `assertAlmostEqual` 210). Run on a `StringIO` stream and print `ran <n>` , `failures <f>` , `errors <e>` , `ok <bool>` , then `failed <name>` per failure, sorted.

Input: the bug name. **Output:** four lines, then any failed names.

```
none
```

prints

```
ran 4
failures 0
errors 0
ok True
```

INPUT 1

```
none
```

OUTPUT 1

```
ran 4
failures 0
errors 0
ok True
```

REFERENCE SOLUTION

```
import io
import unittest

BUG = input().strip()


class BankAccount:
    def __init__(self, balance=0):
        self.balance = balance

    def deposit(self, amount):
        if amount ⩽ 0 and BUG ≠ "negative":
            raise ValueError("amount must be positive")
        self.balance += amount

    def withdraw(self, amount):
        if amount > self.balance and BUG ≠ "overdraw":
            raise ValueError("insufficient funds")
        self.balance -= amount

    def add_interest(self, rate):
        if BUG == "interest":
            self.balance += rate
        else:
            self.balance += self.balance * rate


class TestBankAccount(unittest.TestCase):
    def test_deposit_increases_balance(self):
        account = BankAccount(100)
        account.deposit(50)
        self.assertEqual(account.balance, 150)

    def test_deposit_rejects_non_positive(self):
        account = BankAccount(100)
        with self.assertRaises(ValueError):
            account.deposit(0)
        with self.assertRaises(ValueError):
            account.deposit(-5)

    def test_withdraw_rejects_overdraw(self):
        account = BankAccount(100)
        with self.assertRaises(ValueError):
            account.withdraw(150)

    def test_interest(self):
        account = BankAccount(200)
        account.add_interest(0.05)
        self.assertAlmostEqual(account.balance, 210)
```

```
suite = unittest.defaultTestLoader.loadTestsFromTestCase(TestBankAccount)
result = unittest.TextTestRunner(stream=io.StringIO(), verbosity=0).run(suite)
print("ran", result.testsRun)
print("failures", len(result.failures))
print("errors", len(result.errors))
print("ok", result.wasSuccessful())
for name in sorted(test.id().rsplit(".", 1)[-1] for test, _ in result.failures):
    print("failed", name)
```

EXERCISE 2

Retry against a scripted mock

Write `retry(fn, attempts)` that calls `fn()` up to `attempts` times, returning the first result and swallowing `ConnectionError` between tries; when every attempt fails it raises the last error. Read `attempts` on the first line and a script on the second: tokens that are `fail` (a `ConnectionError("down")`) or an integer (a result). Build `Mock(side_effect=script)`, call `retry` and print `result <value>` or `gave up after <attempts>`, then `calls <mock.call_count>`.

Input: the attempt count, then the script. **Output:** two lines.

```
3
fail fail 42
```

prints

```
result 42
calls 3
```

INPUT 1

```
3
fail fail 42
```

OUTPUT 1

```
result 42
calls 3
```

REFERENCE SOLUTION

```
from unittest.mock import Mock


def retry(fn, attempts):
    last = None
    for _ in range(attempts):
        try:
            return fn()
        except ConnectionError as e:
            last = e
    raise last


attempts = int(input())
script = [ConnectionError("down") if tok == "fail" else int(tok) for tok in input().
fetch = Mock(side_effect=script)
try:
    print("result", retry(fetch, attempts))
except ConnectionError:
    print("gave up after", attempts)
print("calls", fetch.call_count)
```

EXERCISE 3

A counting handler

Subclass `logging.Handler` as `Counting`: its `emit(record)` counts records per `levelname` and keeps `(levelname, message)` for records at `WARNING` or above. Read a threshold level on the first line, then lines `<LEVEL> <message>` until EOF; attach the handler to a logger named `app` (`propagate = False`) set to `DEBUG` so every record reaches it, and emit each line with `logger.log`. Print `counts DEBUG=<n> INFO=<n> WARNING=<n> ERROR=<n> CRITICAL=<n>`, then `at <threshold>: <k> shown` where `k` counts records whose level is at least the threshold, then one line `<LEVEL>: <message>` per kept record in order.

Input: the threshold, then records. **Output:** the counts, the threshold line, the kept records.

```
INFO
DEBUG a
INFO b
WARNING c
ERROR d
```

prints

```
counts DEBUG=1 INFO=1 WARNING=1 ERROR=1 CRITICAL=0
at INFO: 3 shown
WARNING: c
ERROR: d
```

INPUT 1

```
INFO
DEBUG a
INFO b
WARNING c
ERROR d
```

OUTPUT 1

```
counts DEBUG=1 INFO=1 WARNING=1 ERROR=1
CRITICAL=0
at INFO: 3 shown
WARNING: c
ERROR: d
```

REFERENCE SOLUTION

```
import logging
import sys

LEVELS = ["DEBUG", "INFO", "WARNING", "ERROR", "CRITICAL"]


class Counting(logging.Handler):
    def __init__(self):
        super().__init__()
        self.counts = {name: 0 for name in LEVELS}
        self.kept = []

    def emit(self, record):
        self.counts[record.levelname] += 1
        if record.levelno >= logging.WARNING:
            self.kept.append((record.levelname, record.getMessage()))


threshold = sys.stdin.readline().strip()
logger = logging.getLogger("app")
logger.propagate = False
logger.setLevel(logging.DEBUG)
handler = Counting()
logger.addHandler(handler)

for line in sys.stdin:
    level, _, message = line.rstrip().partition(" ")
    if level:
        logger.log(getattr(logging, level), message)

print("counts " + " ".join(f"{name}={handler.counts[name]}" for name in LEVELS))
shown = sum(handler.counts[name] for name in LEVELS if getattr(logging, name) >= get
print(f"at {threshold}: {shown} shown")
for name, message in handler.kept:
    print(f"{name}: {message}")
```

CHECKPOINT QUESTIONS

1. Which test belongs at the bottom of the pyramid, in the greatest number?

A. End-to-end

B. Integration

C. Unit

D. Manual

2. Why one behaviour per test?

A. Style

B. A failure's name is a diagnosis, and one failing assertion does not hide the others

C. Speed

D. unittest requires it

3. Which assertion is right for `0.1 + 0.2` against `0.3`?

A. `assertEqual`

B. `assertAlmostEqual`

C. `assertTrue`

D. `assertIs`

4. `assertRaises(IndexError)` as a context manager and the block returns normally. Result?

A. Pass

B. The test fails — nothing was raised

C. Error

D. Skipped

5. A `subTest` loop has five rows and two fail. How many entries in `result.failures` and how much in `testsRun`?

A. 2 and 5

B. 2 and 1

C. 1 and 1

D. 5 and 5

6. What does a pytest `yield` fixture guarantee that a `return` fixture does not?

A. A value

B. Teardown after the test, even on failure

C. Speed

D. Isolation

7. What is `@pytest.mark.parametrize` mechanically?

A. A loop inside the test

B. Rows attached to the function, each run and reported as its own test

C. A fixture

D. A marker with no effect

8. What does `mock.assert_called_once_with(1, 2)` check?

A. That the mock exists

B. Exactly one call, with arguments `(1, 2)`

C. At least one call

D. That the return value is `(1, 2)`

9. A patch has no effect on the code under test. The most likely cause?

A. Mocks are disabled

B. The target names where the function is defined, not where the module under test looks it up

C. The test is async

D. `spec` was used

10. Which is a fake rather than a mock?

A. `Mock(return_value=3)`

B. An in-memory dict-backed class with the repository's methods

C. `patch("x.y")`

D. `assert_called`

11. Which log level threshold shows WARNING and ERROR but not INFO?

A. `DEBUG`

B. `INFO`

C. `WARNING`

D. `CRITICAL`

12. After a crash, what does `pdb.pm()` give you?

A. A fresh run under the debugger

B. The frames of the exception that just happened, with their local variables

C. A log file

D. A profile

ANSWER KEY · CHECKPOINT — TESTING AND DEBUGGING

1. C Unit tests are fastest and point at the failing line.

2. B A test checking five unrelated things stops at the first failure.

3. B Floats compare by rounded difference.

4. B That is how a suite catches a function that stopped raising.

5. B The method is one test; each failing subtest is reported separately.

6. B The runner resumes the generator after the test; a returned value has no after.

7. B Small enough to rebuild in plain Python when pytest is absent.

8. B `assert_called_with` checks only the last call; `assert_any_call` any call.

9. B Patch the name in the module under test.

10. B A fake works; a mock records and asserts interactions.

11. C A record passes when its level is at least the threshold.

12. B Post-mortem inspection at the point of failure — faster than adding breakpoints.