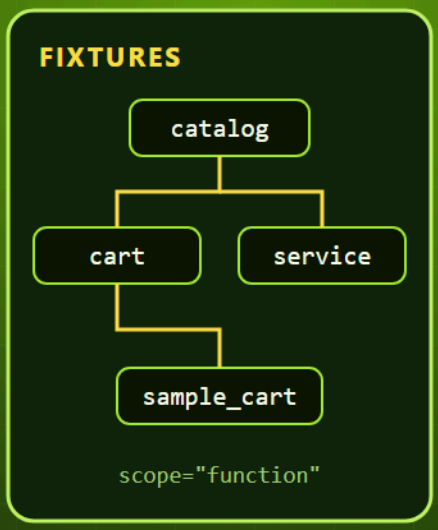
tests/test_pricing.py

```python
@pytest.mark.parametrize(
  "code, total",
  [("GLOW10", "63.22"),
   ("WICK5", "65.25")],
)
def test_total(sample_cart, code, total):
  p = price_cart(sample_cart, "US", ...)
  assert p.total == Decimal(total)
```




```
$ python -m pytest -q
.................................................. [ 49%]
.................................................. [100%]
105 passed in 2.65s
```

ALL GREEN

A COMPLETE, PRACTICAL GUIDE

Testing Python with pytest

Fixtures, parametrization, mocking, property-based testing, and test suites you can trust

assert · fixtures · conftest · parametrize · markers
unittest.mock · Hypothesis · async · coverage · CI

PYTHON 3.12+ · PYTEST 9 · REAL, RUNNABLE EXAMPLES

THENEXBOOKS

Testing Python with pytest

Fixtures, Parametrization, Mocking, Property-Based Testing, and Test Suites You Can Trust
From your first assert to a fast, reliable suite running in continuous integration

First Edition

THENEXBOOKS

Testing Python with pytest
Fixtures, Parametrization, Mocking, Property-Based Testing, and Test Suites You Can Trust

First Edition, 2026

Copyright © 2026 thenexbooks (thenexbooks.gumroad.com). All rights reserved.

No part of this publication may be reproduced, distributed, or transmitted in any form or by any means without the prior written permission of the publisher, except for brief quotations in reviews and certain other non-commercial uses permitted by copyright law. Code examples and configuration files may be used freely in your own projects.

The information in this book is provided "as is" without warranty of any kind. While every precaution has been taken in its preparation, the publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein. Python, pytest, and the libraries and services described in this book change over time; command-line options, configuration settings, output formats, and default behavior can differ between versions. Always consult the official documentation for the versions you use.

Python is a registered trademark of the Python Software Foundation. GitHub and GitHub Actions are trademarks of GitHub, Inc. Windows is a trademark of the Microsoft group of companies. pytest, Hypothesis, pytest-asyncio, HTTPX, SQLAlchemy, mypy, Ruff, coverage.py, and other projects named in this book are the property of their respective owners and maintainers. Their use in this book is for identification only and does not imply endorsement. This book is independent and is not affiliated with, sponsored by, or endorsed by the Python Software Foundation, the pytest project, or any other project or company mentioned.

Tallowbrook Candle Co., the online candle shop used as the running example in this book, is fictional, as are its products, coupons, orders, and all sample data. The carrier rate service and its web address use a reserved example domain and do not exist.

Preface

Most Python developers start testing the same way: a script that prints a few values, a glance at the output, and a quiet hope that nothing breaks next week. It works until the code grows. A small change to the discount rules breaks shipping, a refactor that "only moved things around" changes a total by one cent, and a release goes out on Friday because nobody had time to click through every case by hand. Fear of change creeps in, and the codebase starts to set like cooling wax.

Automated tests are how professional teams keep code soft enough to reshape. pytest makes writing them pleasant: a test is a plain function with a plain `assert`, and pytest takes care of finding it, running it, and explaining clearly what went wrong. Behind that simple surface sits a remarkably deep toolkit — fixtures for setup and teardown, parametrization for whole tables of cases, markers for organizing large suites, and a plugin system that has grown an entire ecosystem.

This book teaches pytest from the very first test to a suite you can trust in production. We do not stop at the syntax. We build and test one realistic codebase step by step, introduce each technique at the moment it solves a problem we actually have, and explain *why* it works the way it does — so that you can design good tests for your own code, not just copy patterns.

Who This Book Is For

- **Python developers who have written few or no tests** and want a practical, thorough path from "it works on my machine" to a reliable test suite.
- **Developers who already use pytest by copying examples** and want to understand fixtures, scopes, conftest files, parametrization, and plugins properly.
- **Backend, data, and tooling engineers** who need to test code that talks to databases, web APIs, the clock, the file system, and the command line.
- **Team leads and reviewers** who want a shared vocabulary for test design, flaky tests, coverage, and continuous integration.

You should be comfortable writing Python functions, classes, and modules. No previous experience with testing is assumed.

What You Will Learn

- What to test and why, how to design code that is easy to test, and how to read pytest's output and tracebacks with confidence.
- Assertions in depth: exceptions, warnings, approximate values, rich failure diffs, and helper functions that fail clearly.
- Fixtures from first principles: setup and teardown with `yield`, scopes, `conftest.py`, factory fixtures, and test data builders.

- Parametrization and markers: tables of cases, readable test IDs, skip and xfail, custom markers, and selecting exactly the tests you need.
- Isolation: temporary files, environment variables, captured output and logs, `unittest.mock` with autospec — and when a simple fake beats a mock.
- Hard-to-test code: time, randomness, HTTP clients with `httpx.MockTransport`, SQLite and SQLAlchemy databases, async code with pytest-asyncio, and command-line programs.
- Property-based testing with Hypothesis, which finds the cases you did not think of and shrinks them to a minimal example.
- Quality and workflow: coverage, pytest configuration and plugins, flaky tests, test-driven development, type checking and linting, continuous integration with GitHub Actions, and keeping a growing suite healthy.

How This Book Is Organized

Part I — pytest Foundations (Chapters 1–5) covers why we test, writing and running tests, assertions, and designing testable code. **Part II — Fixtures and Parametrization** (Chapters 6–10) covers fixtures, scopes, conftest files, factories and builders, parametrization, and markers. **Part III — Isolation and Test Doubles** (Chapters 11–15) covers temporary files, monkeypatching, captured output and logs, mocking, fakes, time and randomness, and HTTP code. **Part IV — Beyond Unit Tests** (Chapters 16–19) covers databases, property-based testing, async code, and command-line programs. **Part V — Quality, Tooling, and Workflow** (Chapters 20–24) covers coverage, configuration and plugins, flaky and slow tests, test-driven development, and static checks. **Part VI — Test Suites in Practice** (Chapters 25–28) covers continuous integration, suite maintenance, integration and end-to-end tests, and a complete case study. The appendices provide a pytest cheat sheet, templates and reference tables, a production checklist for test suites, and a glossary.

Each chapter includes exercises under **Try It Yourself** and ends with a **Key Takeaways** box for quick review.

The Running Example: Tallowbrook Candle Co.

Throughout the book we test the Python code behind **Tallowbrook Candle Co.**, a fictional small online shop that sells hand-poured jar candles, tins, pillars, tealights, and accessories. Its shop core manages a catalog with stock levels, shopping carts, coupons such as `GLOW10` and `WICK5`, weight-based shipping with a free-shipping threshold, checkout and order numbers, a client for a shipping carrier's rate API, an order database, a command-line tool, CSV exports for the bookkeeper, gift card codes, and Candle Club loyalty points. The rules are small enough to check by hand and rich enough to contain every classic testing problem: money rounding, boundaries, dates and deadlines, randomness, network failures, and databases.

A complete reference project accompanies the book: the `tallowbrook` package and a test suite of more than a hundred tests, all of which pass. Every test and every piece of output in this book was produced by running the code with real Python and real pytest.

Versions and Conventions

Examples target **Python 3.12** or later and **pytest 9**. Nearly everything also works with pytest 8; where a feature is new in pytest 9, the text says so. Other libraries — pytest-asyncio, Hypothesis, HTTPX, SQLAlchemy 2, mypy, and Ruff — are used in their recent versions, and Notes point out where older tutorials differ. Commands are shown as `python -m pytest`; on Windows you can type `py -m pytest`. Test timings in the output depend on your machine and will differ from the book, and file paths are shown as `C:\work\tallowbrook`.

> **TIP**
>
> A recommended practice or a technique that makes your work easier.

> **NOTE**
>
> Background information, version differences, or a useful aside.

> **WARNING**
>
> A common mistake or something that can cause false confidence, flaky tests, or lost work.

Code appears in `monospace`. Each listing is labeled with its file name, such as `tests/test_cart.py`; commands are labeled **Terminal** and their results **Output**. Long lines are wrapped to fit the page.

Contents

Preface

PART I — PYTEST FOUNDATIONS

1 Why Test? What to Test, and What pytest Gives You

2 Your First Tests: Installing pytest, Writing Tests, and Plain assert

3 Running pytest: Discovery, Layout, Selection, and Reading Failures

4 Assertions in Depth: Exceptions, Warnings, Approximate Values, and Clear Failures

5 Designing Testable Code: Pure Functions, Seams, and the Tallowbrook Core

PART II — FIXTURES AND PARAMETRIZATION

6 Fixture Fundamentals: Setup, Teardown, and yield

7 Fixture Scope, conftest.py, and Sharing Fixtures

8 Factory Fixtures, Fixture Composition, and Test Data Builders

9 Parametrization: One Test, Many Cases

10 Markers: skip, xfail, Custom Markers, and Selecting Tests

PART III — ISOLATION AND TEST DOUBLES

11 Files, Environment, Output, and Logs: tmp_path, monkeypatch, capsys, and caplog

12 Mocking with unittest.mock: Mock, patch, and autospec

13 When Not to Mock: Fakes, Stubs, and Dependency Injection

14 Testing Time and Randomness

15 Testing HTTP Code with httpx MockTransport

PART IV — BEYOND UNIT TESTS

16 Database Tests with SQLite and SQLAlchemy

17 Property-Based Testing with Hypothesis

18 Async Tests with pytest-asyncio

19 Testing Command-Line Programs

PART V — QUALITY, TOOLING, AND WORKFLOW

20 Coverage: Measuring What Your Tests Exercise

21 Configuring pytest: pyproject.toml, Plugins, and Hooks

22 Flaky Tests and Fast Suites: Isolation, Ordering, and Speed

23 Test-Driven Development: A Walkthrough

24 Type Checking and Linting Alongside Tests: mypy and ruff

PART VI — TEST SUITES IN PRACTICE

25 Continuous Integration with GitHub Actions

26 Maintaining a Test Suite: Smells, Refactoring, and Legacy Code

27 Integration and End-to-End Tests: Wiring the Whole Checkout

28 Case Study: The Complete Tallowbrook Test Suite

APPENDICES

A pytest Cheat Sheet

B Templates and Reference

C Test Suite Production Checklist

D Glossary

PART I

pytest Foundations

Why automated tests matter and what is worth testing, writing and running your first pytest tests, finding and selecting tests, reading failures with confidence, assertions in depth, and shaping code so it is easy to test.

1 · Why Test?
2 · Your First Tests
3 · Running pytest
4 · Assertions in Depth
5 · Designing Testable Code

CHAPTER 1 · PART I — PYTEST FOUNDATIONS

Why Test? What to Test, and What pytest Gives You

Before we write a single fixture or marker, we need a clear picture of what automated tests are for, what they cannot do, and where to aim them first. In this chapter you meet Tallowbrook Candle Co., compare three ways of writing the same test in Python, and run your first pytest test against real shop code.

A Small Change With a Large Blast Radius

Imagine a quiet Tuesday morning at Tallowbrook Candle Co., a small online shop that hand-pours jar candles, tins, pillars, and tealights in its own workshop. The shop runs on a modest Python package: a product catalog, a cart, coupons, a shipping calculator, and a checkout that turns a cart into an order. A developer on the Tallowbrook team is tidying up the money helpers and notices this function:

src/tallowbrook/money.py (excerpt)

```
def round_money(amount: Decimal) -> Decimal:
    """Round to the cent, half-up (7.025 -> 7.03)."""
    return amount.quantize(CENT, rounding=ROUND_HALF_UP)
```

It looks a little fussy. Python has a built-in `round()`, so why not write `return round(amount, 2)` and drop the import of `ROUND_HALF_UP`? The change is one line, it reads more naturally, and every quick check in the interactive prompt looks fine: `round(Decimal("12.5"), 2)` gives `Decimal('12.50')`, and `round(Decimal("9.751"), 2)` gives `Decimal('9.75')`.

But `round()` on a `Decimal` uses the context's default rounding mode, which is *round half to even* (often called banker's rounding). For the value `7.025` it gives `7.02`, not `7.03`. That value is not hypothetical: a 10% coupon on a $70.25 cart is exactly $7.025, and Tallowbrook's published rule is that money rounds half-up to the cent. After the "harmless" cleanup, a customer with the GLOW10 coupon would be charged one cent too little, the order total would no longer match the receipt the shop's bookkeeper reconciles, and nobody would notice until the numbers stopped adding up weeks later.

Now imagine the same change in a project with a three-line test:

tests/test_first.py

```
from decimal import Decimal

from tallowbrook.money import round_money
```

```
def test_round_money_rounds_half_up() -> None:
    assert round_money(Decimal("7.025")) == Decimal("7.03")
```

We made the one-line "cleanup" in a scratch copy of the project and ran the test:

Terminal

```
python -m pytest -q tests/test_first.py
```

Output

```
F                                                                    [100%]
================================= FAILURES =================================
___________________________ test_round_money_rounds_half_up ___________________________

    def test_round_money_rounds_half_up() -> None:
>       assert round_money(Decimal("7.025")) == Decimal("7.03")
E       AssertionError: assert Decimal('7.02') == Decimal('7.03')
E        +  where Decimal('7.02') = round_money(Decimal('7.025'))
E        +    where Decimal('7.025') = Decimal('7.025')
E        +  and   Decimal('7.03') = Decimal('7.03')

tests\test_first.py:7: AssertionError
========================== short test summary info ==========================
FAILED tests/test_first.py::test_round_money_rounds_half_up - AssertionError:...
1 failed in 7.39s
```

Within seconds the mistake is visible, together with the exact value the function returned (`Decimal('7.02')`) and the value we expected. The developer reverts the line, perhaps adds a comment explaining why `ROUND_HALF_UP` matters, and moves on. That is the whole case for automated testing in miniature: a small, cheap check that runs every time, remembers a rule so humans do not have to, and speaks up the moment the rule is broken.

> **NOTE**
>
> Timings on your machine will differ from the ones shown in this book. The first run in a fresh folder is often slower because Python compiles modules and pytest rewrites assertions and caches the result. We keep the real timings in the output but you should never read anything into them.

What Automated Tests Buy You

A test is code that runs other code and checks the result. That simple definition hides several different benefits, and it helps to name them, because each one pushes you toward writing a slightly different kind of test.

Confidence to change code

The biggest payoff is not catching bugs in code you just wrote; it is letting you change code later without fear. Software that cannot be changed safely becomes software nobody wants to touch. Teams then work around old code instead of fixing it, and every workaround makes the next change harder. A good test suite turns "I think this refactor is fine" into "the 105 checks that describe how the shop behaves still pass." It does not make the change risk-free, but it moves the risk from production into your terminal, where it is cheap.

Executable documentation

Comments and wiki pages drift away from the code over time. Tests cannot drift silently, because a test that no longer matches the code fails. A test named `test_coupon_valid_on_its_last_day` tells a new team member something about the coupon rules that the code itself only implies: the expiry date is inclusive. When someone asks "what happens if a customer orders exactly $60.00 of candles?", the answer is a test they can read and run, not a recollection.

Design feedback

Code that is hard to test is usually hard to use. If testing the checkout requires a real database, a real clock, and a real network call to the carrier, that is information: the checkout is tangled with things it should merely use. Writing tests early exposes those tangles while they are still easy to fix. Chapter 5 is devoted to this idea, and you will see how Tallowbrook's design (an injected clock, an injected repository, an injected HTTP client) grew out of it.

Faster debugging

When something breaks, a focused failing test points at the problem far more precisely than a bug report saying "the total was wrong on Tuesday." Once you can reproduce a bug as a test, you can fix it with a tight feedback loop, and the test stays behind as a guard so the same bug cannot return quietly. Bugs that come back are called *regressions*, and the tests that prevent them are *regression tests*.

What Tests Cannot Do

It is just as important to be honest about the limits. A passing test suite does not prove that your program is correct. It shows that the specific examples you thought of behave the way you expected. If nobody wrote a test for an international order heavier than twenty kilograms, the suite says nothing about that case. Testing can reveal the presence of bugs, but a green run is not evidence of their absence.

Tests also inherit the mistakes of the people who write them. If the developer believes shipping is free from $50.00 and writes the test that way, the code and the test can agree with each other and both be wrong. This is why tests should be grounded in a written rule or a real example (an invoice, a ticket, a product decision) rather than in whatever the code currently returns.

Finally, tests cost something. They take time to write, time to run, and time to maintain when behavior changes on purpose. A test that checks every private detail of an implementation breaks on every refactor and teaches the team to ignore failures. Part of the craft we teach in this book is writing tests that fail *only* when something that matters has changed.

> **WARNING**
>
> Never write the assertion by copying whatever the code returns today. If you run the function, see `7.02`, and paste `7.02` into the test, you have frozen the bug in place. Work out the expected value from the rule first, and only then compare it with the output.

Meet Tallowbrook Candle Co.

Every chapter in this book tests the same codebase, so it is worth getting to know it now. Tallowbrook Candle Co. is a fictional shop, but its rules are the kind real shops have: prices in dollars and cents, limited stock, coupons with conditions, and shipping that depends on weight and destination. Here is the sample catalog the reference project ships with:

SKU	Name	Price	Grams	Stock
TB-101	Cedar Smoke Jar 8 oz	$24.00	340	25
TB-102	Lemon Verbena Tin 4 oz	$12.50	180	40
TB-103	Lavender Field Pillar	$18.00	450	12
TB-104	Sea Salt Orchid Jar 16 oz	$38.00	680	4
TB-105	Unscented Tealights 12-pack	$9.75	220	60
TB-106	Balsam Fir Three-Wick	$46.00	1050	0 (sold out)
TB-107	Brass Wick Trimmer	$14.00	90	15

Each product also has a category (jar, tin, pillar, tealight, or accessory). Anything with five or fewer units in stock counts as low stock, which today means TB-104 and TB-106. The shop accepts four coupons:

Coupon	Kind	Value	Min subtotal	Last valid day
GLOW10	percent	10%	$0.00	2026-12-31
WICK5	fixed	$5.00	$25.00	2026-12-31
SHIPFREE	free shipping	-	$30.00	2026-12-31
WINTER25	percent	25%	$50.00	2026-02-28

The book's "today" is Tuesday, March 10, 2026, at 09:30 UTC, so WINTER25 has already expired. Coupon codes are case-insensitive, a discount can never exceed the subtotal, and every discount is rounded half-up to the cent.

Shipping is where most of the interesting edge cases live. A parcel to the United States costs $5.95 for the first 500 grams plus $1.25 for each additional *started* 500 grams; any other country costs $18.00 plus $4.50 per started 500 grams. US orders ship free when the subtotal after the discount is $60.00 or more, or when the customer uses SHIPFREE. A parcel must weigh between 1 and 20,000 grams. The total is always subtotal minus discount plus shipping.

The worked example you will see again and again is a cart with two Cedar Smoke Jars, one Lemon Verbena Tin, and one pack of tealights. Its subtotal is $70.25 and it weighs 1,080 grams. With GLOW10 the discount is $7.03 (half-up from $7.025), the discounted subtotal of $63.22 clears the free-shipping threshold, and the total is $63.22. The reference project includes a small command-line tool, so you can see that calculation for yourself:

Terminal

```
python -m tallowbrook quote TB-101:2 TB-102 TB-105 --coupon GLOW10
```

Output

```
  2 x Cedar Smoke Jar 8 oz           $48.00
  1 x Lemon Verbena Tin 4 oz         $12.50
  1 x Unscented Tealights 12-pack     $9.75
Subtotal                            $70.25
Discount                            -$7.03
Shipping                             $0.00
Total                               $63.22
```

(The command needs the package on Python's import path. From the project root you can set `PYTHONPATH=src` for the command, or install the project in editable mode as shown in Chapter 2.) A smaller order, one pillar and one wick trimmer, shows the other side of the threshold:

Terminal

```
python -m tallowbrook quote TB-103 TB-107
```

Output

```
  1 x Lavender Field Pillar          $18.00
  1 x Brass Wick Trimmer             $14.00
Subtotal                             $32.00
Discount                              $0.00
Shipping                              $7.20
Total                                $39.20
```

The parcel weighs 540 grams: 500 grams at $5.95 plus one started block of 500 grams at $1.25 gives $7.20. If you can follow that arithmetic, you already know enough about the domain to read every test in this book.

Behind the command line sits a package of small modules: `money`, `catalog`, `cart`, `discounts`, `shipping`, `pricing`, `orders`, a `clock` abstraction, a SQLite-backed order repository in `db`, an HTTP client for a fake carrier rate service in `carrier`, CSV export, gift codes, and Candle Club loyalty points. Chapter 5 tours them in detail. For now, the important point is that each rule above is a candidate for a test.

Kinds of Tests

People sort tests into categories, and the names are useful as long as you do not treat them as laws. Three categories cover most of what you will write:

Kind	What it exercises	Tallowbrook example
Unit test	One function or class, in memory, no I/O	`shipping_cost(540, "US", ...)` returns $7.20
Integration test	Several parts together, or code plus a real resource	Save an order to SQLite and read it back
End-to-end test	The whole program from the outside	Run the CLI in a subprocess and read its output

Unit tests are fast and precise: when one fails, you know which function misbehaved. Integration tests catch the bugs that live in the gaps between parts, such as a timestamp that loses its time zone on the way into a database (a real lesson waiting for you in Chapter 16). End-to-end tests prove that the pieces are actually wired together, but they are slower, and when one fails it can take a while to find out why.

You will often see these categories drawn as shapes. The *test pyramid* suggests a wide base of unit tests, fewer integration tests, and a small tip of end-to-end tests. The *testing trophy* argues for putting most of the effort into integration tests, because that is where it says the confidence is. Both are opinions, shaped by the kinds of systems their authors worked on. A pure calculation library may be almost all unit tests; a thin web service that mostly glues a database to an API may get more value from integration tests. In this book the Tallowbrook suite ends up with many fast unit tests for the money rules, a solid layer of database and HTTP tests, and a handful of end-to-end checkout and CLI tests. We will explain each choice when we make it rather than follow a shape.

> **TIP**
>
> Instead of asking "is this a unit test or an integration test?", ask two practical questions: "how fast is it?" and "when it fails, how quickly can I tell what broke?" A test that is fast and points precisely at the problem is valuable whatever you call it.

What to Test First, and What Not to Test

A brand-new project, or an old one with no tests, can feel overwhelming. You cannot test everything at once, and you should not try. Start where a bug would hurt most and where the rules are sharpest.

Start with money and rules

Anything that calculates what a customer pays deserves tests first. At Tallowbrook that means `round_money`, discounts, shipping, and the price breakdown. These functions take values and return values, so they are also the easiest to test: no files, no network, no clock.

Test the boundaries

Bugs cluster at edges. The free-shipping rule says "$60.00 or more," so the interesting inputs are $59.99 and $60.00, not $10 and $200. A parcel of 500 grams costs $5.95 but 501 grams costs $7.20. WICK5 needs at least $25.00, so $24.99 must be rejected and $25.00 accepted. WINTER25 is valid *on* February 28. One of the most instructive cases in the whole project is the cart worth $60.50 with GLOW10: the discount of $6.05 drops it to $54.45, below the threshold, so shipping suddenly costs $7.20 and the total is $61.65. Without the coupon, the customer pays $60.50 with free shipping. A coupon that makes an order more expensive is surprising, and surprising behavior is exactly what tests should pin down, whether the business decides to keep it or change it.

Test the errors

The unhappy paths matter as much as the happy ones. What happens when someone adds a sold-out product, asks for 25 of something when the limit per line is 24, types an unknown coupon, or enters `USA` instead of a two-letter country code? Each of these has a defined behavior in Tallowbrook (a specific exception with a specific message), and each deserves a test. Error handling is code that runs rarely, which means it is code that breaks quietly.

What not to test

- **Other people's code.** You do not need to test that `Decimal` adds correctly or that SQLAlchemy can insert a row. Test *your* use of them: that your repository stores and returns an order faithfully.
- **Trivial code with no logic.** A dataclass field that is stored and read back is not worth a dedicated test. It will be exercised by every test that uses the object.
- **Implementation details.** Test what a function promises (inputs, outputs, errors, effects), not how it does the work internally. If you can rewrite a function's body without changing its behavior and a test fails, the test was checking the wrong thing.
- **Everything twice.** If ten tests already exercise the cart subtotal through different paths, an eleventh adds maintenance cost without adding much confidence.

Three Ways to Write the Same Test

Python gives you more than one way to write automated tests. Before we commit to pytest, let us write the half-up rounding check three ways and run each one. That makes it clear what pytest adds and why the rest of the book uses it.

unittest: the standard library framework

The `unittest` module ships with Python. It follows the xUnit tradition found in many languages: you write a class that inherits from `unittest.TestCase`, put tests in methods whose names start with `test`, and check results with assertion methods such as `assertEqual`.

tests/test_money_unittest.py

```
import unittest
from decimal import Decimal

from tallowbrook.money import round_money


class RoundMoneyTests(unittest.TestCase):
    def test_rounds_half_up(self) -> None:
        self.assertEqual(round_money(Decimal("7.025")), Decimal("7.03"))


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

Terminal

```
python -m unittest -v tests.test_money_unittest
```

Output

```
test_rounds_half_up (tests.test_money_unittest.RoundMoneyTests.test_rounds_half_up)
    ... ok

----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

(We wrapped the first output line to fit the page; on screen it is a single line.) This works, and `unittest` is a perfectly respectable tool with no installation required. The costs are ceremony and vocabulary: a class for every group of tests, `self.` in front of every check, and a long list of assertion methods to remember (`assertEqual`, `assertIn`, `assertAlmostEqual`, `assertRaises`, `assertCountEqual`, and many more). Setup and cleanup go into `setUp` and `tearDown` methods, which are tied to the class hierarchy and hard to share flexibly.

doctest: examples inside documentation

The `doctest` module, also in the standard library, finds text that looks like an interactive Python session and checks that running it produces the output shown. It is a lovely way to keep examples in documentation honest:

docs/money.txt

```
Rounding money at Tallowbrook
=============================

>>> from decimal import Decimal
>>> from tallowbrook.money import round_money
>>> round_money(Decimal("7.025"))
Decimal('7.03')
```

Terminal

```
python -m doctest -v docs/money.txt
```

Output

```
Trying:
    from decimal import Decimal
Expecting nothing
ok
Trying:
    from tallowbrook.money import round_money
Expecting nothing
ok
Trying:
    round_money(Decimal("7.025"))
Expecting:
    Decimal('7.03')
ok
1 items passed all tests:
   3 tests in money.txt
3 tests in 1 items.
3 passed and 0 failed.
Test passed.
```

Doctests compare the *printed text* of results, which is both their charm and their weakness. They are excellent for keeping README examples and docstrings correct, but awkward for anything with setup, for output that varies (dictionary ordering in older code, memory addresses, timestamps), or for checking many cases. Treat them as documentation that happens to be tested, not as your main test suite.

pytest: plain functions and plain assert

Here is the same check written for pytest, the file you saw at the start of the chapter:

tests/test_first.py

```
from decimal import Decimal

from tallowbrook.money import round_money


def test_round_money_rounds_half_up() -> None:
    assert round_money(Decimal("7.025")) == Decimal("7.03")
```

Terminal

```
python -m pytest tests/test_first.py
```

Output

```
============================= test session starts =============================
platform win32 -- Python 3.12.0, pytest-9.1.1, pluggy-1.6.0
rootdir: C:\work\tallowbrook
configfile: pyproject.toml
plugins: anyio-4.15.1, hypothesis-6.168.5, asyncio-1.4.0
asyncio: mode=Mode.STRICT, debug=False, asyncio_default_fixture_loop_scope=function,
  asyncio_default_test_loop_scope=function
collected 1 item

tests\test_first.py .                                                    [100%]

============================== 1 passed in 4.49s ==============================
```

The test is a module-level function whose name starts with `test_`. There is no class, no import from pytest, and no special assertion method: just Python's own `assert` statement. When the assertion fails, pytest inspects the expression and shows you the values on both sides, as you saw in the banker's-rounding failure earlier. That feature is called *assertion rewriting*, and Chapters 2 and 4 explain how it works.

Do not worry about the header lines yet; Chapter 2 walks through each of them. The plugin list in particular depends on what is installed in your environment, so yours may be shorter or longer.

pytest runs the others too

Choosing pytest does not mean throwing away existing tests. pytest collects and runs `unittest.TestCase` classes as they are, and it can collect doctests from text files or modules:

Terminal

```
python -m pytest -q tests/test_money_unittest.py
python -m pytest -q --doctest-glob="*.txt" docs/money.txt
```

Output

```
.                                                                        [100%]
1 passed in 1.94s
.                                                                        [100%]
1 passed in 3.48s
```

This is a practical advantage when you inherit a project. You can switch the test *runner* to pytest on day one, keep every old `unittest` test working, and write new tests in the pytest style. Over time, as you touch old tests, you can convert them, or leave them alone if they work.

> **NOTE**
> pytest is a third-party package, not part of the standard library, so you install it into each project's environment (Chapter 2 shows how). It is widely used across the Python ecosystem, actively maintained, and backed by a large collection of plugins. This book targets pytest 9; almost everything also works on pytest 8, and we point out the few features that need pytest 9 and later.

What pytest Gives You

If pytest only removed the class and the `self.assertEqual` ceremony, it would be a pleasant convenience. Its real strength is a set of ideas that scale from one test to thousands. Here is a preview of the toolkit, with the chapter where each piece is covered properly:

Feature	What it does for you	Chapter
Plain `assert`	Readable checks with detailed failure diffs	2, 4
Discovery and selection	Find tests by name; run one, some, or the failures	3
`pytest.raises`, `approx`	Check exceptions, warnings, and floats	4
Fixtures	Setup and teardown by asking for it by name	6-8
Parametrization	One test function, many input cases	9
Markers	Skip, expect failure, label, and select tests	10
Built-in fixtures	Temp files, environment, captured output, logs	11
Plugins and hooks	Extend pytest itself with a few functions	21

The single most important idea in that table is the *fixture*. In pytest, a test declares what it needs by naming it as a parameter, and pytest provides it. A test that needs a cart with the worked example in it simply asks for `sample_cart`: