AHMED ADAWY TECH CAPSULES

Python Testing with pytest â€” Free Sample

A Practical Guide to Writing Reliable Tests

Author	Ahmed Adawy
Category	Python / Testing
Difficulty	Intermediate
Language	English
Version	1.0
Release	2026

Professional Technical Capsule

Table of Contents

Python Testing with pytest

A Practical Guide to Writing Reliable Tests

Ahmed Adawy Tech Capsules

Introduction

Software becomes harder to maintain as it grows.

A small Python project may begin with only a few functions. At that stage, manually checking whether everything works can seem reasonable.

You change a function.

You run the program.

You check the result.

Everything looks fine.

But as the project grows, manual verification becomes increasingly expensive.

A change in one function can unexpectedly break another part of the application.

A refactoring can introduce a regression.

A new feature can work correctly while silently breaking an older feature.

This is where automated testing becomes valuable.

Automated tests provide a repeatable way to verify that software behaves as expected.

Instead of relying entirely on manual checking, developers can create a collection of tests that execute automatically.

In the Python ecosystem, `pytest` is one of the most practical tools for writing and running automated tests.

This capsule focuses on using pytest to build reliable tests for Python projects.

1. Why Automated Testing Matters

Testing is not simply about finding bugs.

A good test suite also provides confidence when software changes.

Consider a simple function:

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

A basic test could verify its expected behavior:

```
def test_add():
    assert add(2, 3) == 5
```

This test establishes a simple contract:

As the project grows, more tests can protect more behavior.

For example:

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


def divide(a, b):
    if b == 0:
        raise ValueError("Cannot divide by zero")


def is_positive(value):
    return value > 0
```

The corresponding tests can verify normal behavior as well as invalid input:

```python
def test_add():
    assert add(2, 3) == 5


def test_divide():
    assert divide(10, 2) == 5


def test_is_positive():
    assert is_positive(10) is True
```

Every test describes an expected behavior.

That behavior becomes part of the project's safety net.

2. What pytest Provides

`pytest` is a Python testing framework designed to make writing and running tests simple.

A typical testing workflow looks like this:

```
Write code
    â†“
Write tests
    â†“
Run pytest
    â†“
Inspect failures
    â†“
Fix code
    â†“
Run pytest again
```

This cycle can be repeated whenever the project changes.

The important advantage is that the developer does not need to manually verify every behavior after every modification.

The test suite performs those checks consistently.

A simple pytest test can be written as:

```python
def test_addition():
    result = 2 + 3
    assert result == 5
```

The `assert` statement checks whether the actual result matches the expected result.

If the condition is true, the test passes.

If the condition is false, pytest reports the failure.

3. Installing pytest

The simplest way to install pytest is with pip:

```
pip install pytest
```

For a project that uses a virtual environment, activate the environment first.

Then install pytest:

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

You can verify the installation with:

```
pytest --version
```

4. Your First pytest Test

A common project structure separates application code from test code.

```
project/
â”œâ”€â”€ src/
â”‚   â””â”€â”€ calculator.py
â”œâ”€â”€ tests/
â”‚   â””â”€â”€ test_calculator.py
â”œâ”€â”€ requirements.txt
â””â”€â”€ pyproject.toml
```

Application code lives inside `src/`.

Test code lives inside `tests/`.

A simple calculator module might contain:

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

The test file could contain:

```
from calculator import add


def test_add():
    assert add(2, 3) == 5
```

The function name begins with `test_`.

This naming convention allows pytest to discover the test automatically.

Run the tests with:

```
pytest
```

A successful run confirms that the tested behavior currently matches the expected result.

5. Assertions

Assertions are at the heart of many pytest tests.

An assertion expresses something that should be true.

For example:

```
assert 2 + 3 == 5
```

Or:

```
result = add(10, 5)

assert result == 15
```

You can also test boolean behavior:

```
def is_positive(value):
    return value > 0
```

The test can be:

```
def test_is_positive():
    assert is_positive(10) is True
```

And:

```
def test_negative_number():
    assert is_positive(-5) is False
```

The purpose of an assertion is not merely to calculate a result.

It defines the expected behavior of the software.

6. Testing Normal Behavior

The first type of behavior we usually test is normal, expected input.

Consider:

```
def multiply(a, b):
    return a * b
```

A simple test is:

```
def test_multiply():
    assert multiply(4, 5) == 20
```

We can test several behaviors independently:

```
def test_multiply_positive_numbers():
    assert multiply(4, 5) == 20


def test_multiply_by_zero():
    assert multiply(10, 0) == 0


def test_multiply_negative_number():
    assert multiply(-3, 4) == -12
```

Each test communicates one specific expectation.

This makes failures easier to understand.

7. Testing Edge Cases

Normal input is only part of testing.

Real software also receives unusual or boundary values.

Consider:

```
def percentage(value, total):
    return (value / total) * 100
```

A basic test could be:

```
def test_percentage():
    assert percentage(25, 100) == 25
```

But we should also think about values such as:

- Zero
- Negative numbers
- Very large numbers
- Empty values
- Boundary values

The exact edge cases depend on the behavior of the function.

The important question is:

Those inputs deserve explicit tests.

8. Testing Exceptions

Software should also behave predictably when invalid input is provided.

Consider a division function:

```
def divide(a, b):
    if b == 0:
        raise ValueError("Cannot divide by zero")

    return a / b
```

Testing the normal case is straightforward:

```
def test_divide():
    assert divide(10, 2) == 5
```

But the invalid case is equally important.

pytest provides a convenient way to verify that an exception is raised:

```
import pytest


def test_divide_by_zero():
    with pytest.raises(ValueError):
        divide(10, 0)
```

This test checks that the function fails in the expected way.

9. Testing Exception Messages

Sometimes checking the exception type is not enough.

The error message itself may be part of the expected behavior.

```
import pytest


def test_divide_by_zero_message():
    with pytest.raises(ValueError,
match="Cannot divide by zero"):
        divide(10, 0)
```

Now the test verifies both:

- The expected exception type.
- The expected message.

10. Keeping Tests Focused

A growing project needs a predictable testing structure.

A test should ideally verify one behavior or a closely related group of behaviors.

Avoid creating enormous tests that verify many unrelated things.

Instead of:

```
def test_everything():
    ...
```

prefer focused tests:

```
def test_user_creation():
    ...

def test_user_email_validation():
    ...

def test_user_permissions():
    ...
```

Focused tests are easier to read.

They are easier to debug.

They also provide better feedback when something fails.

Good test names are part of good documentation.

11. Arrange, Act, Assert

A useful structure for many tests is:

```
Arrange
   â†“
Act
   â†“
Assert
```

For example:

```python
def test_add():
    # Arrange
    a = 2
    b = 3

    # Act
    result = add(a, b)

    # Assert
    assert result == 5
```

Arrange

Prepare the data and objects required by the test.

Act

Execute the behavior being tested.

Assert

Verify that the result matches the expectation.

This structure is not mandatory, but it makes many tests easier to understand.

12. Running pytest

Once tests are created, they can be executed with:

```
pytest
```

You can also run a specific test file:

```
pytest tests/test_calculator.py
```

Or a specific test:

```
pytest tests/test_calculator.py::test_add
```

This becomes particularly useful when debugging a failing test.

13. Reading Test Results

A successful pytest run typically reports that tests passed.

For example:

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

tests/
test_calculator.py ....
[100%]

============================== 4 passed in 0.05s
==============================
```

A failure is reported differently:

```
================================== FAILURES
==================================
_____________________________ test_multiply
_____________________________

    def test_multiply():
>       assert multiply(4, 5) == 25
E       assert 20 == 25
```

The failure output tells us:

- Which test failed.
- Which assertion failed.
- The actual value.
- The expected value.

This makes automated testing useful not only for verification but also for debugging.

14. A Small Real-World Example

Consider a tiny utility module:

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


def divide(a, b):
    if b == 0:
        raise ValueError("Cannot divide by zero")

    return a / b


def is_positive(value):
    return value > 0
```

A corresponding test file could be:

```
import pytest

from calculator import add
from calculator import divide
from calculator import is_positive


def test_add():
    assert add(2, 3) == 5


def test_divide():
    assert divide(10, 2) == 5


def test_divide_by_zero():
    with pytest.raises(ValueError):
        divide(10, 0)


def test_is_positive():
    assert is_positive(10) is True
```

This small example demonstrates:

- Normal function testing
- Assertions
- Exception testing
- Focused test functions
- Readable test names

Run it with:

```
pytest
```

15. What We Have Learned

At this point, we have established the foundation of automated testing with pytest.

We have seen how to:

- Install pytest
- Create test files
- Write test functions
- Use assertions
- Test normal behavior
- Test edge cases
- Test exceptions
- Verify exception messages
- Organize focused tests
- Follow Arrange, Act, Assert
- Run individual tests
- Read pytest output

These ideas form the foundation for everything that follows.

But a professional test suite needs more than individual test functions.

As applications become larger, repeating the same setup and test patterns becomes inefficient.

That is where pytest's more powerful features become useful.

16. Where the Full Capsule Goes Next

The complete **45-page** capsule continues beyond this introduction.

The next sections cover:

- Fixtures
- Reusable test setup
- Parametrization
- Testing multiple input combinations
- Advanced exception testing
- Test organization
- Code coverage
- Continuous integration
- Testing real-world projects
- Professional testing practices

These features help transform a collection of simple tests into a maintainable automated testing system.

About the Full Capsule

Python Testing with pytest

A Practical Guide to Writing Reliable Tests

Author: Ahmed Adawy

Category: Python / Testing

Difficulty: Intermediate

Language: English

Version: 1.0

The complete capsule contains approximately **45 pages** of practical material.

It is designed for Python developers who want to understand automated testing and apply pytest to real projects.

The free sample intentionally stops before the more advanced sections.

Thank You for Reading

Thank you for reading this free sample of **Ahmed Adawy Tech Capsules**.

If you found the material useful, the complete capsule continues with the practical techniques required to build a reliable pytest testing workflow.

Keep testing. Keep improving. Keep building.

Ahmed Adawy

Technical Author â€¢ Python Developer â€¢ AI Educator

Ahmed Adawy Tech Capsules

Practical engineering knowledge, one capsule at a time.