Pytest

pytest is a powerful and flexible testing framework for Python that simplifies writing and running tests. It provides an intuitive syntax, detailed reporting, and advanced features like fixtures, parameterization, and plugins. pytest is widely used in small projects and enterprise applications because it allows developers to write concise, readable tests with minimal boilerplate.

Unlike Python's built-in unittest module, which follows a more rigid class-based structure, pytest enables writing tests using plain functions and assert statements, making test writing more natural and efficient.

This chapter will contain the following topics:

- Writing basic tests using plain functions and assert statements
- Installing pytest and running tests from the command line
- Understanding why automated testing matters in real projects
- Using fixtures for setup and cleanup (@pytest.fixture)
- Parameterizing tests with different inputs (@pytest.mark.parametrize)
- Interpreting test results: passed, failed, skipped
- Organizing test files and naming conventions for discovery
- Skipping tests and handling expected failures with markers
- Mocking and patching code for isolated testing (pytest-mock, monkeypatch)
- Measuring test coverage with pytest-cov
- Running tests in parallel with pytest-xdist
- Debugging tests interactively using --pdb
- Tagging and filtering tests using custom markers
- Integrating pytest into CI/CD pipelines (e.g. GitHub Actions)
- Following best practices for maintainable, scalable test suites

Why Automated Testing is Important

Automated testing is a fundamental practice in software development that ensures code correctness, reliability, and maintainability. Manual testing is time-consuming and prone to human error, making it ineffective for large projects or frequent code changes. Automated tests allow developers to:

- **Catch bugs early**: Prevent regressions when modifying code.
- **Improve code quality**: Encourage better design by enforcing testability.
- **Speed up development**: Automate repetitive checks, reducing manual testing time.
- **Enable continuous integration (CI)**: Ensure stability before deploying changes.

- **Increase confidence in refactoring**: Modify code without fear of breaking existing functionality.

In modern software development, automated testing is a necessity rather than an option, and pytest makes it easy to integrate testing into development workflows.

Overview of pytest Features

Simple and Readable Tests: Write tests as plain functions with assert statements.

Fixtures for Setup and Teardown: Reusable setup and teardown logic with @pytest.fixture.

Parametrized Tests: Run a test with multiple sets of inputs:

```
import pytest


@pytest.mark.parametrize("num, expected", [(1, 2), (2, 4), (3, 6)])
def test_double(num, expected):
    assert num * 2 == expected
```

Powerful Markers: Skip or expect failures in tests:

```
@pytest.mark.skip(reason="Not implemented yet")
def test_future_feature():
    pass
```

Parallel Test Execution: Run tests in parallel using pytest-xdist:

```
pytest -n 4
```

Rich CLI Options: Run specific tests:

```
pytest test_example.py::test_function
```

- Stop on first failure:

```
pytest -x
```

- Show verbose output:

```
pytest -v
```

Great IDE Support: Works seamlessly with VS Code, PyCharm, and other editors.

pytest simplifies testing in Python by offering a more intuitive, powerful, and extensible framework compared to unittest. Its features, such as **fixtures, assertions, parameterized tests, and plugins**, make it the preferred choice for modern Python testing.

In the next chapter, we will explore **installing and running pytest**, including setting up tests and understanding test output.

Installing and Running pytest

pytest is a widely used testing framework in Python because of its simplicity and powerful features. In this chapter, we will explore how to install pytest, run tests in a project, understand test output, and execute multiple tests across a directory.

Installing pytest Using pip

Installing pytest is straightforward using pip, the standard package manager for Python.

Installation Command

To install pytest, run the following command:

```
pip install pytest
```

To verify the installation and check the installed version:

```
pytest --version
```

Example output:

```
pytest 7.4.0
```

Installing pytest in a Virtual Environment

It is best practice to install pytest inside a virtual environment to avoid dependency conflicts.

macOS/Linux:

```
python -m venv venv
source venv/bin/activate  # On macOS/Linux
```

Windows:

```
venv\Scripts\activate
pip install pytest
```

Once installed, pytest is ready to run tests in your project.

Running Tests in a Project

Once pytest is installed, you can execute tests easily. The recommended approach is to create a **test file** that follows pytest's naming conventions.

Running Tests in a Single File

To run tests in a specific script, use:

```
pytest test_math.py
```

Running All Tests in a Directory

To run all test files in a directory, navigate to the project folder and run:

```
pytest
```

This will automatically discover and run all tests in files matching **test_*.py** or ***_test.py**.

Example Project Structure

my_project/

```
├── tests/
│   ├── test_math.py
│   ├── test_string.py
│   └── test_database.py
└── my_script.py
```

Running pytest in the tests/ folder executes all tests inside test_math.py, test_string.py, and test_database.py.

Running a Specific Test Function

To run a specific test function, use:

```
pytest test_math.py::test_addition
```

This isolates execution to only test_addition() in test_math.py.

Running Tests with Verbose Output

To get detailed test output, use:

```
pytest -v
```

Example output:

```
test_math.py::test_addition PASSED
test_math.py::test_subtraction FAILED
```

Conclusion

- pytest is installed using pip install pytest.
- Tests are written as functions prefixed with test_.
- Run tests with pytest test_script.py or pytest for all tests.
- pytest automatically discovers test files.
- Failure reports provide detailed debugging information.

In the next chapter, we will dive deeper into writing test functions and using assertions effectively in pytest.

Writing Your First Test with pytest

Testing is an essential part of software development, ensuring that your code behaves as expected. In Python, **pytest** is one of the most widely used testing frameworks due to its simplicity and powerful features. In this section, you'll learn how to write and run your first test using pytest.

Basic Test Function (Assert Statements)

A basic test function in pytest is simply a regular Python function that contains one or more **assert** statements. The assert statement is used to check whether a condition holds true. If the condition is false, the test fails; otherwise, it passes.

Here's an example of a simple test function:

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

How It Works:

1. The function name starts with test_, which is a convention pytest recognizes.
2. The assert statement checks whether 2 + 3 equals 5.
3. If the condition holds true, pytest marks the test as **passed**; otherwise, it fails.

Using Multiple Assertions

You can include multiple assertions within a single test function:

```
def test_math_operations():
    assert 2 * 3 == 6
    assert 10 - 5 == 5
    assert 8 / 2 == 4
```

Each assert statement is evaluated independently. If one fails, pytest will report the failure but still evaluate the others.

Naming Conventions for Test Functions

Pytest automatically discovers and runs test functions based on specific naming conventions:

- Test function names **must start with** test_ (e.g., test_example()).
- Test file names should also begin with test_ (e.g., test_math.py).
- If using test classes, the class name should start with Test (e.g., class TestMathOperations:).

Examples of good test function names:

```
def test_string_length():
```

```
    assert len("pytest") == 6


def test_list_append():
    my_list = []
    my_list.append(1)
    assert my_list == [1]
```

Running a Test and Interpreting Results

Once you've written your test, you can run it using pytest.

Running Tests

To execute your test, navigate to the directory containing your test file and run:

```
pytest
```

Or, to run a specific test file:

```
pytest test_math.py
```

Pytest will discover all test functions automatically and execute them.

Interpreting the Output

When pytest runs, it provides output indicating which tests passed or failed:

✅ Successful Test Output

```
============================= test session starts
==============================

collected 1 item


test_math.py .
[100%]


============================== 1 passed in 0.02s
==============================
```

- The **dot (.)** indicates a test passed.
- The [100%] shows test progress.
- The last line confirms the number of tests that passed.

❌ Failed Test Output

If a test fails, pytest provides a detailed failure report:

```
============================= test session starts
==============================
```

```
collected 1 item


test_math.py F
[100%]


================================= FAILURES
=====================================

_____________________________ test_addition
_____________________________


    def test_addition():
>       assert 2 + 3 == 6
E       AssertionError: assert 5 == 6


test_math.py:2: AssertionError
========================== short test summary info
==========================
FAILED test_math.py::test_addition - AssertionError: assert 5 == 6
========================== 1 failed in 0.02s
============================
```

- The F indicates a test **failed**.
- The error message AssertionError: assert 5 == 6 tells us what went wrong.
- The line number (test_math.py:2) helps locate the issue.

Summary

- Write test functions using the assert statement.
- Follow pytest naming conventions (test_ prefix for functions and files).
- Run tests using pytest in the terminal.
- Interpret pytest results to identify passing and failing tests.

By following these steps, you've taken your first step toward robust testing using pytest.

Using pytest Fixtures for Test Setup and Cleanup

Writing clean and maintainable tests often requires setting up preconditions (such as initializing objects, connecting to databases, or preparing test data) and cleaning up afterward. **pytest fixtures** provide a powerful way to manage such setup and teardown operations in an organized and reusable manner.

What Are Fixtures?

A **fixture** in pytest is a function that provides **setup and cleanup code** for tests. Instead of repeating setup code in every test, fixtures allow you to define it once and reuse it across multiple test functions.

Why Use Fixtures?

- **Eliminates redundant setup code** in test functions.
- **Ensures clean test states** by handling setup and teardown systematically.
- **Improves test maintainability** by keeping setup logic separate from test logic.
- **Can be shared across multiple test files**.

Creating and Using Fixtures with @pytest.fixture

To define a fixture, use the @pytest.fixture decorator. The fixture function is then passed as an argument to test functions that require it.

Example: Simple Fixture

```
import pytest


@pytest.fixture
def sample_data():
    return {"name": "pytest", "version": 1.0}


def test_data_content(sample_data):
    assert sample_data["name"] == "pytest"
    assert sample_data["version"] == 1.0
```

How It Works:

1. The @pytest.fixture decorator marks sample_data as a fixture.
2. The test function test_data_content() accepts sample_data as an argument.
3. When pytest runs, it **injects** the fixture return value into the test.

Using Fixtures for Setup and Cleanup

Some tests require **setup before** and **cleanup after** execution. You can use yield inside a fixture to define cleanup logic.

Example: Using yield for Cleanup

```
import pytest


```

```
@pytest.fixture
def file_handler():
    # Setup: Create a temporary file
    file = open("test_file.txt", "w")
    file.write("Hello, pytest!")
    file.close()

    yield "test_file.txt"  # Provide test access to the file

    # Cleanup: Remove the file after test
    import os
    os.remove("test_file.txt")

def test_file_content(file_handler):
    with open(file_handler, "r") as f:
        content = f.read()
    assert content == "Hello, pytest!"
```

How It Works:

1. The fixture file_handler() creates a temporary file before the test runs.
2. The yield statement provides the test function access to the file.
3. After the test completes, **code after yield executes**, removing the file.

Using autouse=True for Implicit Setup

By default, fixtures must be **explicitly included** in test function arguments. However, by setting autouse=True, a fixture will automatically execute **for every test in the module**.

Example: Automatic Database Setup

```
import pytest

@pytest.fixture(autouse=True)
def setup_database():
    print("\nSetting up database...")
    yield
    print("\nTearing down database...")
```

```
def test_user_creation():
    print("Testing user creation")
    assert True

def test_user_deletion():
    print("Testing user deletion")
    assert True
```

Output:

```
Setting up database...
Testing user creation
Tearing down database...

Setting up database...
Testing user deletion
Tearing down database...
```

When to Use autouse=True

- When a fixture is needed for **all tests** in a module.
- When setup and teardown should happen automatically **without passing the fixture explicitly**.

Sharing Fixtures Across Multiple Test Files

If you need to use the same fixture across multiple test files, define it in a separate conftest.py file. pytest automatically discovers fixtures in conftest.py and makes them available without imports.

Example: Defining Fixtures in conftest.py

```
import pytest0

@pytest.fixture
def api_client():
    return {"base_url": "https://api.example.com", "auth_token": "abc123"}
```

Using the Fixture in Multiple Test Files

```
# test_auth.py
```

```
def test_login(api_client):
    assert api_client["base_url"] == "https://api.example.com"
```

```
# test_data.py
def test_fetch_data(api_client):
    assert "auth_token" in api_client
```

Why Use conftest.py?

- No need to import fixtures in every test file.
- Keeps test files **clean and modular**.
- Makes fixtures available **globally** across test modules.

Summary

- **Fixtures** help with **test setup and cleanup**, making tests more organized.
- Use @pytest.fixture to define reusable test setup logic.
- Use yield inside fixtures to handle **cleanup operations**.
- Use autouse=True to automatically apply a fixture to **all tests** in a module.
- Store shared fixtures in **conftest.py** for use across multiple test files.

With pytest fixtures, you can create structured and maintainable test environments.

Parametrizing Tests with pytest.mark.parametrize

When writing tests, it is often necessary to run the same test logic with multiple sets of inputs. Instead of writing repetitive test functions, **pytest provides @pytest.mark.parametrize**, which allows running a test function multiple times with different inputs.

Running a Test with Multiple Inputs

A common scenario in testing is verifying that a function produces correct outputs for various inputs. Instead of duplicating test functions, you can use **@pytest.mark.parametrize** to test multiple cases efficiently.

Example: Testing a Function with Different Inputs

Let's say we have a function that squares a number:

```
def square(x):
    return x * x
```

Instead of writing separate test functions, we can use @pytest.mark.parametrize:

```
import pytest
```

```
@pytest.mark.parametrize("input_value, expected_output", [
    (2, 4),
    (3, 9),
    (4, 16),
    (5, 25)
])
def test_square(input_value, expected_output):
    assert square(input_value) == expected_output
```

How It Works:

1. The @pytest.mark.parametrize decorator takes two arguments:
    - A **string of variable names** (comma-separated).
    - A **list of tuples**, each containing a test case (input, expected output).
2. The test function is executed **once per tuple**, injecting the values dynamically.
3. If any test case fails, pytest reports the specific input that caused the failure.

Defining Test Cases Dynamically

For more complex scenarios, parameterization can handle multiple variables, allowing dynamic test cases.

Example: Testing String Reversal with Multiple Cases

```
def reverse_string(s):
    return s[::-1]


@pytest.mark.parametrize("input_str, expected_str", [
    ("hello", "olleh"),
    ("pytest", "tsetyp"),
    ("12345", "54321"),
    ("", "")  # Testing an edge case (empty string)
])
def test_reverse_string(input_str, expected_str):
    assert reverse_string(input_str) == expected_str
```

Benefits of This Approach:

- Reduces duplication by testing multiple cases in a **single test function**.
- Makes test cases **easy to modify and expand**.

- Provides clear failure reports when a test case fails.

Improving Test Coverage with Parameterization

Parameterization is useful for **testing edge cases** and **boundary values**, improving overall test coverage.

Example: Testing a Function with Edge Cases

Let's test a division function, ensuring it handles:

- **Regular division**
- **Zero as the numerator**
- **Negative values**
- **Zero division error**

```
def safe_divide(a, b):
    if b == 0:
        return None  # Avoiding ZeroDivisionError
    return a / b


@pytest.mark.parametrize("a, b, expected", [
    (10, 2, 5),
    (0, 5, 0),  # Zero numerator
    (-10, 2, -5),  # Negative division
    (5, 0, None)  # Zero denominator (handled gracefully)
])
def test_safe_divide(a, b, expected):
    assert safe_divide(a, b) == expected
```

How It Improves Test Coverage:

- **Tests normal and edge cases** (zero values, negative numbers).
- **Prevents division by zero errors** by explicitly handling the case.
- **Catches unexpected behavior early** in development.

Summary

- Use @pytest.mark.parametrize to **run a test function multiple times** with different inputs.
- Parameterized tests **reduce code duplication** and make test maintenance easier.
- They **improve test coverage** by handling edge cases dynamically.

- Multiple parameters can be tested by passing **tuples of values**.

Using test parameterization allows for **efficient, scalable, and maintainable** test cases

Handling Expected Failures and Exceptions

While testing, there are scenarios where you expect a function to raise an exception or a test to fail under certain conditions. Pytest provides powerful tools to handle such cases gracefully using:

- **pytest.raises** for testing expected exceptions.
- **pytest.mark.xfail** for marking tests that are expected to fail.
- **pytest.mark.skip and pytest.mark.skipif** for conditionally skipping tests.

Testing for Expected Exceptions with pytest.raises

Sometimes, a function should raise an exception for invalid inputs. **pytest.raises** allows us to verify that an exception occurs as expected.

Example: Testing a Division Function That Raises an Exception

```
import pytest


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


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

How It Works:

1. The test uses pytest.raises(ZeroDivisionError) inside a with block.
2. If divide(10, 0) raises ZeroDivisionError, the test **passes**.
3. If no exception or a different exception is raised, the test **fails**.

Checking Exception Messages

To validate the exception message, use match= inside pytest.raises:

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

```
        divide(10, 0)
```

This ensures the raised exception contains the correct message.

Marking Tests as Expected Failures with pytest.mark.xfail

Sometimes, you may have tests that are expected to fail due to a known bug or incomplete implementation. Instead of removing or commenting them out, you can mark them with **pytest.mark.xfail**.

Example: Expected Failure Due to a Bug

```
import pytest


def buggy_function():
    return 2 * 2  # Expected: 5, but returns 4 (incorrect)


@pytest.mark.xfail
def test_buggy_function():
    assert buggy_function() == 5  # This test will fail
```

How It Works:

1. The test is **expected to fail**, so pytest reports it as an XFAIL (expected failure).
2. If the test unexpectedly **passes**, pytest reports it as XPASS (unexpected pass).
3. The test does **not count as a failure**, keeping the test suite clean.

Conditionally Failing Tests

You can conditionally mark a test as expected to fail only on specific platforms or versions:

```
@pytest.mark.xfail(condition=(sys.platform == "win32"), reason="Bug
on Windows")
def test_windows_bug():
    assert buggy_function() == 5
```

This test will **only be marked as XFAIL on Windows**, but will run normally on other platforms.

Skipping Tests Based on Conditions with pytest.mark.skip and pytest.mark.skipif

Skipping Tests Unconditionally with pytest.mark.skip

If a test should always be skipped (e.g., deprecated functionality or long-running tests), use @pytest.mark.skip.

```
@pytest.mark.skip(reason="Skipping this test for now")
```

```
def test_skipped():
    assert 1 + 1 == 2  # This test will not run
```

Pytest will **not execute** this test and will show it as **SKIPPED** in the test report.

Skipping Tests Conditionally with pytest.mark.skipif

Sometimes, a test should be skipped **only under certain conditions**, such as:

- When a required dependency is missing.
- When running on a specific operating system.
- When using a certain Python version.

```
import sys
import pytest


@pytest.mark.skipif(sys.version_info < (3, 8), reason="Requires
Python 3.8+")
def test_new_python_feature():
    assert some_python_3_8_feature() == "Expected Result"
```

How It Works:

1. The test **runs only if Python 3.8 or newer** is used.
2. If Python is older, pytest **skips the test** and reports it as SKIPPED.

Summary

- Use pytest.raises to test for expected exceptions.
- Use pytest.mark.xfail to mark tests that are **expected to fail** (e.g., due to known bugs).
- Use pytest.mark.skip to **skip tests unconditionally**.
- Use pytest.mark.skipif to **conditionally skip tests** based on environment conditions.

These tools help in writing **robust, controlled, and maintainable** test suites.

Using Markers and Customizing Test Execution

In **pytest**, markers provide a flexible way to categorize, filter, and control the execution of tests. They allow you to organize your test suite and run specific groups of tests based on tags, making test execution more efficient and manageable.

What Are pytest Markers?

Markers are special labels that can be applied to test functions to group them logically. These labels can then be used to selectively execute or skip tests during test runs. Pytest

includes built-in markers like @pytest.mark.skip and @pytest.mark.xfail, but you can also define custom markers for your specific testing needs.

Example of a built-in marker:

```
import pytest


@pytest.mark.skip(reason="Skipping this test temporarily")
def test_example():
    assert 1 + 1 == 2
```

Running Specific Test Groups with pytest -m

Markers are useful when you want to run only a subset of tests instead of the entire suite. The -m flag in pytest allows you to specify a marker and execute only the tests associated with it.

For example, suppose you have tests marked as slow and fast:

```
import pytest


@pytest.mark.slow
def test_heavy_computation():
    result = sum(range(1000000))
    assert result > 0


@pytest.mark.fast
def test_quick_addition():
    assert 2 + 2 == 4
```

To run only the tests marked as slow, use the following command:

```
pytest -m slow
```

Similarly, you can exclude tests by using not:

```
pytest -m "not slow"
```

Creating and Using Custom Markers

You can define your own custom markers to better organize your tests. However, pytest requires custom markers to be registered in pytest.ini to avoid warnings.

Example:

```
import pytest
```

```
@pytest.mark.api
def test_api_response():
    response = {"status": 200}
    assert response["status"] == 200
```

To run only tests marked as api, use:

```
pytest -m api
```

Configuring Markers in pytest.ini

To avoid warnings about unregistered markers, you should list custom markers in the pytest.ini file:

```
[pytest]
markers =
    slow: Marks tests as slow
    fast: Marks tests as fast
    api: Marks API-related tests
```

By configuring markers in pytest.ini, you ensure better test organization and maintainability while preventing pytest from showing unnecessary warnings.

Structuring and Organizing Tests

A well-structured test suite makes debugging, maintaining, and scaling your test cases easier. Following best practices ensures clarity and efficiency when running and managing tests.

Organizing Test Files and Directories

When structuring your test suite, consider the following standard directory layout:

```
project_root/
│── src/                # Your application code
│── tests/              # Test suite
│    ├── __init__.py     # (optional) Makes the directory a
package
│    ├── test_module1.py # Test file for module1
│    ├── test_module2.py # Test file for module2
│    ├── conftest.py     # Shared fixtures/configuration
```

```
│    ├──── integration/    # Folder for integration tests
│    └──── unit/           # Folder for unit tests
```

Key principles for organizing tests:

- **Use a dedicated tests/ directory** to keep tests separate from application code.
- **Prefix test files with test_** so pytest can automatically discover them.
- **Group related tests into subdirectories** like unit/ and integration/.
- **Keep individual test cases small and focused**, with one assertion per test where possible.

Using conftest.py for Shared Fixtures

conftest.py is a special pytest file that allows you to define **shared fixtures, hooks, and configuration** without needing explicit imports in each test file. It helps **reduce code duplication** and keeps test files cleaner.

Example: Define a shared fixture in conftest.py:

```
import pytest


@pytest.fixture
def sample_data():
    return {"name": "Test", "age": 25}
```

Now, any test in the **same directory or its subdirectories** can use sample_data without needing to import it:

```
def test_sample_data(sample_data):
    assert sample_data["age"] == 25
```

Running Specific Test Modules or Classes

You can run specific test files, modules, or classes using pytest command-line arguments.

Run all tests in a specific module

```
pytest tests/test_module1.py
```

Run a specific test function

```
pytest tests/test_module1.py::test_function_name
```

Run all tests in a specific class

```
pytest tests/test_module1.py::TestClass
```

Run all tests in a directory (e.g., unit tests)

```
pytest tests/unit/
```

Using these commands allows you to **target specific test cases** quickly, improving efficiency when debugging or running focused test groups.

Capturing and Testing Output with capfd and caplog

When writing tests, it's often necessary to **capture and verify printed output or log messages**. Pytest provides built-in fixtures like **capfd** (for capturing standard output) and **caplog** (for verifying log messages) to facilitate this.

Capturing Printed Output with capfd

The capfd fixture captures data sent to **stdout (standard output) and stderr (standard error)**, allowing you to assert that expected output was printed.

Example:

```
import pytest


def greet():
    print("Hello, pytest!")


def test_greet(capfd):
    greet()
    captured = capfd.readouterr()  # Capture output
    assert captured.out.strip() == "Hello, pytest"
```

How capfd.readouterr() works:

- captured.out contains **stdout** (regular print output).
- captured.err contains **stderr** (error messages printed with sys.stderr).

Testing Standard Error Output

```
import sys


def print_error():
    print("An error occurred!", file=sys.stderr)


def test_print_error(capfd):
    print_error()
    captured = capfd.readouterr()
```

```
    assert captured.err.strip() == "An error occurred!"
```

Verifying Log Messages with caplog

The caplog fixture captures **log messages** from the Python logging module, allowing you to test if certain logs were generated.

Example:

```
import logging


def log_message():
    logger = logging.getLogger(__name__)
    logger.warning("This is a warning message")


def test_log_message(caplog):
    with caplog.at_level(logging.WARNING):
        log_message()
    assert "This is a warning message" in caplog.text
```

Additional caplog Features:

Checking the number of log **entries**

```
assert len(caplog.records) == 1  # Ensures only one log was captured
```

Checking log levels

```
assert caplog.records[0].levelname == "WARNING"
```

Filtering logs by logger name

```
assert caplog.records[0].name == "my_logger"
```

Using capfd and caplog helps ensure your application prints the expected messages and logs, making debugging and test validation more efficient.

Mocking and Patching with pytest-mock

Mocking is a crucial technique in testing, allowing you to **replace real objects, functions, or external dependencies** with controlled substitutes. This ensures that tests run in **isolation** without making actual API calls, modifying databases, or relying on external services.

Using unittest.mock with pytest

Python's unittest.mock module provides tools like Mock and patch to replace dependencies during tests. The pytest-mock plugin simplifies working with unittest.mock by providing a mocker fixture.

Example: Mocking a function:

```
from unittest.mock import Mock


def get_user_name(user_id):
    # Simulating a database call
    return f"User-{user_id}"


def test_get_user_name(mocker):
    mock_func = mocker.patch("__main__.get_user_name",
return_value="MockUser")
    assert get_user_name(1) == "MockUser"
    mock_func.assert_called_once_with(1)  # Verify it was called
correctly
```

Mocking External API Calls

When testing functions that call external APIs, **mocking** prevents unnecessary HTTP requests and speeds up tests.

Example: Mocking an API call using requests.get:

```
import requests


def fetch_data():
    response = requests.get("https://api.example.com/data")
    return response.json()


def test_fetch_data(mocker):
    mock_response = mocker.Mock()
    mock_response.json.return_value = {"key": "mocked_value"}
    mocker.patch("requests.get", return_value=mock_response)


    assert fetch_data() == {"key": "mocked_value"}
```

Here, we **patch** requests.get to return a **mock response**, ensuring that no actual HTTP request is made.

Patching Functions and Classes with monkeypatch

Pytest's monkeypatch fixture allows modifying or replacing attributes, functions, and environment variables **without using unittest.mock**.

Example: Replacing a function at runtime:

```
def get_env_variable():
    import os
    return os.getenv("SECRET_KEY", "default_value")


def test_get_env_variable(monkeypatch):
    monkeypatch.setenv("SECRET_KEY", "mocked_secret")
    assert get_env_variable() == "mocked_secret"
```

Patching Class Methods

```
class Service:
    def get_data(self):
        return "Real Data"


def test_mock_class_method(monkeypatch):
    def mock_get_data(self):
        return "Mocked Data"

    monkeypatch.setattr(Service, "get_data", mock_get_data)

    service = Service()
    assert service.get_data() == "Mocked Data"
```

Using pytest-mock and monkeypatch provides **powerful mocking capabilities**, ensuring your tests run **efficiently and reliably** without unintended side effects.

Measuring Test Coverage with pytest-cov

Test coverage is a critical metric in software testing that helps assess how much of the codebase is executed when tests run. The **pytest-cov** plugin integrates coverage

measurement into pytest, making it easy to generate reports and analyze uncovered code sections. This section covers how to install and use pytest-cov, generate and interpret coverage reports, and maintain high test coverage in your projects.

Installing and Using pytest-cov

The pytest-cov plugin extends pytest by providing coverage analysis powered by coverage.py. To install it, run the following command:

```
pip install pytest-cov
```

Once installed, you can run tests with coverage tracking enabled using:

```
pytest --cov=<module_name>
```

For example, if you have a module called my_project, you can check its test coverage with:

```
pytest --cov=my_project
```

This will output a coverage summary, showing the percentage of covered lines.

If you want to see detailed per-file coverage, you can generate a report using:

```
pytest --cov=my_project --cov-report=term-missing
```

This command highlights lines that were not executed during testing, making it easier to identify untested code.

Generating and Interpreting Coverage Reports

The pytest-cov plugin supports various output formats for coverage reports. Some commonly used options include:

- --cov-report=term-missing → Displays missing lines in the terminal.
- --cov-report=html → Generates an interactive HTML report.
- --cov-report=xml → Produces an XML report, useful for CI/CD pipelines.
- --cov-report=annotate → Creates annotated source files highlighting uncovered lines.

To generate an HTML coverage report:

```
pytest --cov=my_project --cov-report=html
```

After running this command, open the htmlcov/index.html file in a web browser to visually inspect the coverage details.

Interpreting coverage reports involves looking at:

- **Coverage Percentage** → Aim for high coverage, typically above 80%.
- **Missing Lines** → Locate and test unexecuted code paths.
- **Branch Coverage** → Ensure conditional branches (if-else, loops) are adequately tested.

Ensuring High Test Coverage in Projects

Achieving high test coverage involves writing comprehensive test cases that exercise all significant code paths. Some best practices include:

1. **Write Tests for Edge Cases**
   Ensure boundary values and unexpected inputs are tested to verify robustness.
2. **Aim for Both Statement and Branch Coverage**
   Test all lines and conditions within functions to avoid untested logic paths.
3. **Use pytest.mark.parametrize for Efficient Testing**
   Parameterizing test cases allows broader input coverage without redundant test code.
4. **Exclude Non-Critical Files**
   If certain files (e.g., configuration files, migrations) should be excluded, specify them in .coveragerc:
5. [run]
6. omit = my_project/config.py
7. **Integrate Coverage Checks into CI/CD**
   Enforce a minimum coverage threshold using:
8. pytest --cov=my_project --cov-fail-under=80

This ensures that any drop in coverage prevents test success.

Summary

The pytest-cov plugin simplifies test coverage measurement, providing detailed reports on tested and untested code. By leveraging various report formats and setting coverage thresholds, developers can ensure comprehensive test coverage in projects. Maintaining high coverage improves code reliability, reduces regression risks, and enhances software quality over time.

Summary

This chapter introduces **pytest**, a powerful and widely-used testing framework for Python. Unlike Python's built-in unittest module, pytest allows developers to write simple, readable tests using plain functions and assert statements, reducing boilerplate and making test code easier to write and understand. It is especially valued for its flexibility, rich feature set, and excellent support in modern development environments.

The chapter begins by explaining the importance of **automated testing**—how it helps catch bugs early, improves code quality, supports continuous integration, and increases confidence in changes. It then outlines the key features of pytest, such as its clean syntax, fixture system for setup and teardown, support for parameterized tests, and tools for skipping or marking tests as expected to fail.

You also learn how to **install pytest**, run tests using the command line, and understand the structure of test files and functions. The chapter explains how pytest automatically discovers tests based on naming conventions and shows how to interpret test results, including passed, failed, and skipped cases.

Fixtures are covered in depth, highlighting how they help manage setup and cleanup tasks efficiently and how they can be reused across multiple tests. You also explore **test parameterization**, which allows running the same test logic against multiple sets of data, improving both efficiency and test coverage.

The chapter goes on to cover **handling expected failures and exceptions**, demonstrating how pytest makes it easy to check that errors are raised when appropriate. You also learn to use **markers** for organizing tests, running specific groups, and conditionally skipping tests.

For larger projects, the chapter discusses **test organization**, including directory structure and shared fixture management using conftest.py. It also touches on tools for capturing output and logging during tests, as well as mocking dependencies to isolate the logic under test.

Finally, the chapter introduces **plugins and integrations** like pytest-cov for measuring test coverage and pytest-xdist for running tests in parallel. It concludes with best practices for writing clean, maintainable, and scalable test suites, ensuring that pytest becomes an integral part of your development workflow.

Exercise

Here are **10 exercises on pytest**, arranged in **increasing order of difficulty**, designed to reinforce concepts from the chapter:

1. Write Your First Test

Define a function add(a, b) that returns the sum of two numbers. Write a test function using pytest to check that add(2, 3) returns 5.

2. Test Multiple Assertions

Create a function math_ops(x) that returns a tuple (x * 2, x + 3). Write a test that asserts both values are correct when x = 4.

3. Use pytest Fixtures

Write a fixture named sample_user that returns a dictionary like {"name": "Alice", "age": 30}. Write a test that uses this fixture to assert the name and age values.

4. Parameterize a Test

Write a function square(x) and test it using @pytest.mark.parametrize with inputs [(2, 4), (3, 9), (5, 25)].

5. Test for Exceptions

Write a function divide(a, b) that raises a ZeroDivisionError if b == 0. Write a test that asserts the exception is raised using pytest.raises.

6. Use yield in a Fixture for Cleanup

Write a fixture that creates a temporary file before a test and deletes it after. Use yield for teardown and assert in the test that the file was created and then removed.

7. Mock External Behavior

Write a function fetch_api_data() that calls requests.get(). Use pytest-mock to mock the requests.get call and return a fake JSON response in the test.

8. Test with capfd or caplog

Write a function that prints "Success!" and logs a warning. Use capfd to test the printed output and caplog to check that the warning was logged.

9. Mark Tests for Conditional Skipping

Write a test that should only run on Python 3.8 or higher. Use pytest.mark.skipif to skip it otherwise. You can use sys.version_info in the condition.

10. Write a Custom Marker and Run a Subset of Tests

Define a custom marker @pytest.mark.slow and apply it to a long-running test. Register it in pytest.ini. Run only the slow tests using pytest -m slow.