# Introduction to Pytest

Pytest is a testing framework for Python that simplifies writing and running tests. It supports simple unit tests to complex functional testing and integration tests.

## Installation

Install pytest using pip:

pip install pytest

Verify installation:

pytest --version

## Basic Test Structure

* Test file names must start with test\_ or end with \_test.py.
* Test function names must start with test\_.

Example:

# test\_sample.py

def test\_addition():

assert 1 + 1 == 2

def test\_subtraction():

assert 2 - 1 == 1

Run the test:

pytest test\_sample.py

## Assertions

Pytest uses Python's assert statement:

assert expression, "Optional failure message"

Examples:

assert 1 + 1 == 2

assert "pytest" in "Learn pytest framework"

## Test Discovery

Pytest automatically discovers test files and functions.

* To run all tests:

pytest

* To run a specific test file:

pytest test\_sample.py

* To run a specific test function:

pytest test\_sample.py::test\_addition

## Markers

Markers allow categorization of tests.

* **Custom Markers**:

import pytest

@pytest.mark.slow

def test\_long\_running():

assert True

Run tests with a specific marker:

pytest -m slow

* **Built-in Markers**:
  + skip: Skip a test unconditionally.
  + @pytest.mark.skip(reason="Not implemented yet")
  + def test\_skip():

assert False

* + skipif: Skip based on a condition.
  + @pytest.mark.skipif(sys.platform == "win32", reason="Does not run on Windows")
  + def test\_platform():

assert True

* **Register Custom Markers** (Add to pytest.ini):
* [pytest]
* markers =

slow: marks tests as slow (deselect with '-m "not slow"')

## Fixtures

Fixtures provide reusable test setup and teardown functionality.

* **Basic Fixture**:
* import pytest

@pytest.fixture

def setup\_data(): return {"key": "value"}

def test\_using\_fixture(setup\_data): assert setup\_data["key"] == "value"

- \*\*Scope of Fixtures\*\*:

```python

@pytest.fixture(scope="module")

def setup\_module():

# Setup at the module level

pass

Available scopes: function, class, module, package, session.

## Parametrized Tests

Run a test with multiple sets of data.

import pytest

@pytest.mark.parametrize("input,expected", [

(1 + 1, 2),

(2 \* 2, 4),

(3 - 1, 2),

])

def test\_math\_operations(input, expected):

assert input == expected

## Running Tests with Options

* **Run Verbose Mode**: pytest -v
* **Show Test Durations**: pytest --durations=3
* **Stop on First Failure**: pytest -x
* **Run Tests in Parallel** (requires pytest-xdist): pytest -n 4

## Assertions with Plugins

Use the pytest.raises for exception handling:

import pytest

def test\_zero\_division():

with pytest.raises(ZeroDivisionError):

1 / 0

## Output Files and Reports

Generate a detailed report:

pytest --html=report.html

## Common Plugins

1. **pytest-cov**: For code coverage.
2. pip install pytest-cov

pytest --cov=my\_module

1. **pytest-xdist**: For parallel testing.
2. pip install pytest-xdist

pytest -n 4

1. **pytest-html**: Generate HTML reports.
2. pip install pytest-html

pytest --html=report.html

1. **pytest-mock**: Mocking support.

pip install pytest-mock

## Debugging Tests

Run tests with pdb debugging:

pytest --pdb

Use breakpoints in code:

def test\_debug():

breakpoint()

assert 1 == 1

## Best Practices

1. Use meaningful test names.
2. Organize tests into separate files and directories.
3. Use fixtures for reusable setup.
4. Use markers to categorize tests.
5. Parametrize tests to cover multiple cases.
6. Integrate pytest with CI/CD pipelines for automation.

# Page Object Model (POM) in Pytest

The **Page Object Model (POM)** is a design pattern used in automation testing to enhance code readability, maintainability, and reusability. It involves creating a **separate class for each web page** in the application under test, encapsulating the elements and methods specific to that page.

In Pytest, the POM pattern can be implemented effectively, particularly when paired with a tool like **Selenium** for web automation.

**Advantages of POM**

1. **Separation of Concerns**: Keeps the test scripts independent of UI structure changes by centralizing the web element locators.
2. **Reusability**: Common methods (e.g., login) can be reused across multiple test cases.
3. **Improved Readability**: Test scripts focus on what to test, not how to locate elements.
4. **Maintainability**: Changes in UI require updates in the page object classes only, not the test scripts.

**Structure of POM with Pytest**

* 1. **Page Classes**: Define web elements and methods to interact with the web page.
  2. **Test Scripts**: Use page classes to interact with the application.
  3. **Utility Files** (optional): Include reusable code, like configurations, logging, or custom helpers.

**Implementation Example**

## Project Structure

Plaintext ~~~

tests/

├── conftest.py # Pytest configuration and fixtures

├── pages/ # Page object classes

│ ├── login\_page.py

│ ├── home\_page.py

├── test\_login.py # Test script

└── utils/

└── config.py # Configurations

## Step 1: Create Page Object Class

Each page class represents a web page. It contains locators and methods to interact with page elements.

Python ~~~

# pages/login\_page.py

from selenium.webdriver.common.by import By

class LoginPage:

def \_\_init\_\_(self, driver):

self.driver = driver

self.username\_input = (By.ID, "username")

self.password\_input = (By.ID, "password")

self.login\_button = (By.ID, "loginBtn")

def enter\_username(self, username):

self.driver.find\_element(\*self.username\_input).send\_keys(username)

def enter\_password(self, password):

self.driver.find\_element(\*self.password\_input).send\_keys(password)

def click\_login(self):

self.driver.find\_element(\*self.login\_button).click()

## Step 2: Configure Fixtures

Use Pytest fixtures to initialize the browser driver and share it across test cases.

Python ~~~

# conftest.py

import pytest

from selenium import webdriver

@pytest.fixture

def browser():

driver = webdriver.Chrome() # Adjust for your browser

driver.maximize\_window()

yield driver

driver.quit()

## Step 3: Write Test Script

The test script uses the page object classes to perform operations.

Python ~~~

# test\_login.py

from pages.login\_page import LoginPage

def test\_valid\_login(browser):

# Open the login page

browser.get("https://example.com/login")

# Create a LoginPage object

login\_page = LoginPage(browser)

# Perform login

login\_page.enter\_username("valid\_user")

login\_page.enter\_password("valid\_password")

login\_page.click\_login()

# Validate successful login (example)

assert "Dashboard" in browser.title

## Tips for POM with Pytest

1. **Use Config Files**:
   * Store test data, URLs, and browser configurations in a central config.py file.

python

# utils/config.py

BASE\_URL = "https://example.com"

USERNAME = "valid\_user"

PASSWORD = "valid\_password"

1. **Create Base Page Class**:
   * A base class can include common methods like waiting for elements or taking screenshots.

python

# pages/base\_page.py

from selenium.webdriver.support.ui import WebDriverWait

from selenium.webdriver.support import expected\_conditions as EC

class BasePage:

def \_\_init\_\_(self, driver):

self.driver = driver

def wait\_for\_element(self, locator, timeout=10):

return WebDriverWait(self.driver, timeout).until(EC.presence\_of\_element\_located(locator))

1. **Integrate Reporting**:
   * Use plugins like pytest-html to generate test execution reports.

bash

pytest --html=report.html

1. **Modularize Reusable Actions**:
   * If multiple tests require common actions (e.g., login), create helper methods in the page object class.

# Hybrid Framework

A **Hybrid Framework** in PyTest refers to the combination of different types of testing frameworks or approaches to create a more efficient and flexible solution. This framework integrates the best features of multiple testing strategies to handle complex test cases and scenarios. In the context of PyTest, it generally combines:

1. **Data-Driven Testing**: Test data is separated from test logic, and test cases are executed with multiple sets of inputs. This allows for reusability and ensures that the test cases can be run with different data inputs without changing the test code.
2. **Keyword-Driven Testing**: It involves using specific keywords (or actions) that are mapped to functions or test steps. This makes it easier for non-technical users to define and understand test steps while maintaining modularity in the test code.
3. **Behavior-Driven Testing**: Often integrated with tools like **BDD (Behavior-Driven Development)**, it focuses on the behavior of the system being tested. Tests are written in plain language (Gherkin syntax) that explains the system's functionality from the user's perspective.

To construct, build, and develop a **Hybrid Framework** in **PyTest**, you will need to combine various testing strategies such as **data-driven**, **keyword-driven**, and possibly **behavior-driven** testing. Below is a step-by-step guide to help you construct a hybrid framework using PyTest:

## 1. Define the Project Structure

Create a well-organized project structure for your hybrid framework. A sample structure might look like this:

my\_hybrid\_framework/

│

├── tests/

│ ├── test\_login.py

│ ├── test\_registration.py

│ └── test\_search.py

│

├── data/

│ ├── login\_data.json

│ └── registration\_data.json

│

├── utils/

│ ├── helpers.py

│ └── keywords.py

│

├── reports/

│ └── test\_report.html

│

├── config/

│ └── settings.py

│

├── requirements.txt

└── conftest.py

## 2. Install Required Packages

First, make sure to install the necessary dependencies:

bash

pip install pytest pytest-html

## 3. Create Data Files (Data-Driven Approach)

Separate test data from the test logic. You can store data in JSON, CSV, or Excel files.

For example, create a login\_data.json to hold different login credentials:

json

[

{"username": "user1", "password": "pass1", "expected": "Success"},

{"username": "user2", "password": "pass2", "expected": "Failure"}

]

## 4. Create Test Cases (Data-Driven Testing)

Use the @pytest.mark.parametrize decorator to pass the data to the test functions.

Example (test\_login.py):

python

import pytest

import json

# Load test data from the JSON file

def load\_login\_data():

with open("data/login\_data.json") as f:

return json.load(f)

@pytest.mark.parametrize("data", load\_login\_data())

def test\_login(data):

username = data["username"]

password = data["password"]

expected = data["expected"]

# Simulate the login logic

result = login(username, password)

# Assert the result

assert result == expected

def login(username, password):

# A simple mockup of login logic

if username == "user1" and password == "pass1":

return "Success"

else:

return "Failure"

Here:

* **Data-driven testing** is implemented by loading the test data from a JSON file.
* The @pytest.mark.parametrize decorator allows us to run the test for each set of data.

## 5. Create Helper Functions and Keywords (Keyword-Driven Testing)

Helper functions and keywords can be implemented in a separate module to make the code reusable.

*Example (utils/keywords.py):*

python

def login\_to\_application(username, password):

# Implement login logic

if username == "user1" and password == "pass1":

return "Success"

else:

return "Failure"

def logout\_from\_application():

# Implement logout logic

return "Logged out"

*Example (test\_login.py using keywords):*

python

from utils.keywords import login\_to\_application

@pytest.mark.parametrize("data", load\_login\_data())

def test\_login(data):

username = data["username"]

password = data["password"]

expected = data["expected"]

result = login\_to\_application(username, password)

assert result == expected

Here:

* **Keyword-driven testing** is used with a helper function login\_to\_application that encapsulates the login logic.

## 6. Add Behavior-Driven Testing (Optional)

If you're integrating **Behavior-Driven Development (BDD)**, you can use Gherkin-style syntax with the help of pytest-bdd. You can define behavior scenarios in a .feature file and implement steps in Python.

Example (login.feature):

gherkin

Feature: Login functionality

Scenario: Valid user login

Given I am on the login page

When I enter valid credentials

Then I should be redirected to the homepage

Scenario: Invalid user login

Given I am on the login page

When I enter invalid credentials

Then I should see an error message

*Example (test\_login\_bdd.py):*

python

from pytest\_bdd import scenarios, given, when, then

# Define the feature file

scenarios('login.feature')

@given('I am on the login page')

def open\_login\_page():

# Simulate opening the login page

pass

@when('I enter valid credentials')

def enter\_valid\_credentials():

# Simulate entering valid credentials

pass

@then('I should be redirected to the homepage')

def check\_homepage\_redirect():

# Simulate checking redirection

assert True

## 7. Create Fixtures (Setup and Teardown)

Use **fixtures** to set up the test environment (e.g., initializing a browser, database, or login).

Example (conftest.py):

python

import pytest

@pytest.fixture

def setup\_environment():

# Set up the environment (e.g., start the browser or connect to DB)

print("Setting up the environment")

yield

# Clean up (e.g., close the browser or disconnect from DB)

print("Cleaning up the environment")

## 8. Generate Test Reports

You can generate detailed test reports using pytest-html to track test results.

Run PyTest with HTML reporting:

bash

pytest --html=reports/test\_report.html

## 9. Handle Configuration Settings

In the config/settings.py file, you can store configuration variables like browser names, URLs, credentials, etc.

Example:

python

BASE\_URL = "https://myapplication.com"

BROWSER = "chrome"

## 10. Run the Tests

Now that everything is set up, you can run your tests with PyTest:

bash

pytest tests/

## 11. Extend and Maintain the Framework

* **Modularity**: Keep the framework modular so that you can add new test cases, keywords, or data sources easily.
* **Scalability**: As your testing requirements grow, ensure that the framework can scale by adding new features such as parallel execution (using pytest-xdist) or integrating with CI/CD pipelines.

## Summary:

1. **Data-driven testing** for running tests with multiple sets of input data.
2. **Keyword-driven testing** with reusable functions or keywords for common actions.
3. **Behavior-driven testing (BDD)** for user-friendly test scenarios.
4. **Fixtures** for setup and teardown.
5. **Configuration settings** for reusable constants and environment-specific variables.

By combining these strategies, you can create a **flexible, scalable, and maintainable hybrid testing framework** in PyTest that suits different types of test scenarios.