pytest Framework - 1

Pytest is a testing framework that supports features like **unit, integration,** and **functional testing** for automating web applications using Selenium.

Features of pytest

- **Fixtures** ⇒ Provides reusable setup and teardown functions with flexible scopes.
- **Parallel Testing** ⇒ Supports running tests in parallel using plugins like **pytest-xdist.**
- **Skipping Tests** ⇒ Allows tests to be skipped using decorators like **@pytest.mark.skip.**
- **Grouping Tests** ⇒ Organize and group tests using markers for selective execution.
- **Batch Testing** ⇒ Run large test suites efficiently.
- **Parameterization** ⇒ Run the same test with different data using **@pytest.mark.parametrize**.
- **Detailed Reports** ⇒ Generates detailed test reports with clear pass/fail results.

Prerequisite: Install **pytest package** in Project interpreter in pycharm.

Project Structure

- **Project** → **Test suite**(Package/Directory) → **Test cases**(Modules(.py)) → **Test steps**(Test Methods in class)

Naming Conventions in Pytest framework

➔ **Modules** names should start with **test_** or ending with **_test**
◆ **test_login.py** or **login_test.py**
➔ **Class** names should start with "**Test**"
◆ **TestClass**
➔ **Test method** names should start with "**test**"
◆ **testMethod1(self)**
	test_1.py
	`import pytest`
	`class TestClass:`
	`  def testMethod1(self):`
	`  print("this is test method1")`
	`  def testMethod2(self):`
	`  print("this is test method2")`

Ways to execute test case in Pytest

- ➔ Tools → **Python Integrated Tools** → choose pytest as the default test runner.
- ➔ Click on the **green arrow** next to the test function or class. Right-click and choose Run 'pytest for <your test function or class>' from the context menu.
- ➔ **Execute in Terminal -** pytest -v -s package or directory\module name\test method name
  - ◆ **Run single module -** pytest -v -s day1-pytest\test_Login.py
  - ◆ **Run all the modules -** pytest -v -s day1-pytest
  - ◆ **Run specific testMethod in module**
    - pytest -v -s day1-pytest\test_Login.py::TestLogin::test_LoginByEmail

Pytest Fixtures

fixtures are functions that manage the setup and teardown process for test environments. Fixtures allow you to define code that needs to **run before a test (setup)** and **after a test (teardown)**, ensuring a controlled and repeatable test environment. They help in avoiding code duplication, keeping test logic clear, and sharing reusable setup logic across multiple tests.

- **Setup**: This is the preparation phase that occurs before the test runs. It involves allocating resources, setting initial conditions, or configuring external dependencies like databases, files, or networks.

- **Teardown**: This is the cleanup phase that occurs after the test completes, regardless of whether the test passes or fails. It ensures that any resources used during the test are released, cleaned, or reset, such as closing database connections or deleting temporary files.
- Fixture functions are passed as an **argument to testMethod.**
- **fixtures** functions also **returns** value when called which is optional

Scope of fixtures

The scope of a fixture in pytest defines how often the fixture will be created and used in your tests.

1. **Function Scope:** Fixture is called once per test function.
2. **Class Scope:** Fixture is called once per class, shared by all test methods in that class.
3. **Module Scope:** Fixture is called once per module, shared by all test functions in that file.
4. **Session Scope:** Fixture is called once per session, shared by all tests in the test run.

test_2.py (Function Scope)

```
import pytest
@pytest.fixture()  # decorator
def setup():
   print("Launching browser...") #Executes once before every test method
   yield
   print("closing browser..") #Executes Once after every test method
class TestClass:
  def test_Login(self,setup):
     print("This is login test")
  def test_Search(self,setup):
     print("this is search test")
```

test_3.py (class Scope)

```
import pytest
@pytest.fixture(scope = "class")  # decorator
def setup():
  print("Launching browser...")  # Executes once before every class
  yield
  print("closing browser..")  # Executes once after class execution completed
class TestClass:
  def test_Login(self,setup):
     print("This is login test")
  def test_Search(self,setup):
     print("this is search test")
```

test_4.py (module Scope)

```
import pytest
@pytest.fixture(scope = "module")  # decorator
def setup():
  print("Launching browser...")  # Executes once before every module
  yield
```

```
  print("closing browser..") # Executes once after module execution completed
class TestClass:
  def test_Login(self,setup):
    print("This is login test")
  def test_Search(self,setup):
    print("this is search test")
```

test_5.py (session Scope)

```
import pytest
@pytest.fixture(scope = "session")  # decorator
def setup():
  print("Connecting to Database...")  # Executes once before every session
  yield
  print("Disconnecting to Database..")  # Executes once after every session
class TestClass:
  def test_Login(self,setup):
    print("This is login test")
  def test_Search(self,setup):
    print("this is search test")
```

Note

- **Session Scope** is useful when you need to set up something that should last for the entire test run, like a database connection.

Autouse Fixtures

➔ **Autouse Fixtures** can be created to automatically run for all tests without being called in each test, useful for global setup or teardown tasks.

test_6.py (autouse)

```
import pytest
@pytest.fixture(autouse=True)  # decorator
def setup():
  print("Launching browser...")  # Executes before every test method
  yield
  print("closing browser..")  # Executes after every test method
class TestClass:
  def test_Login(self):
    print("This is login test")
  def test_Search(self):
    print("this is search test")
```

Conftest file

➔ For better code maintenance we will specify **Configurations** and **fixtures in conftest.py** and pass as argument in **testMethods** in modules

conftest.py

```
import pytest
```

```
from selenium import webdriver
@pytest.fixture()
def setup():
  options = webdriver.ChromeOptions()
  options.add_experimental_option("detach", True)
  driver = webdriver.Chrome(options=options)
  yield driver  # Provide the driver instance to the test
  driver.quit()  # Ensure the browser is closed after the test
```

test_Login.py

```
from selenium.webdriver.common.by import By
class TestLogin:
  def test_Login(self, setup):
    self.driver = setup
    self.driver.get("https://opensource-demo.orangehrmlive.com/")
    self.driver.implicitly_wait(10)
```

Enter username and password

```
    self.driver.find_element(By.NAME, "username").send_keys("Admin")
    self.driver.find_element(By.NAME, "password").send_keys("admin123")
```

Click the Signin button

```
    self.driver.find_element(By.TAG_NAME, "button").click()
```

Validate login success

```
    try:
      self.status =
self.driver.find_element(By.XPATH,"//h6[normalize-space()='Dashboard1']").is_displayed()
      assert self.status is True
    except:
      assert False
```

Note

- In Pytest by default all the testMethods will be passed unless we put **assertions (validation point)**.
- Pytest **automatically discovers** and loads the `conftest.py` file for configurations and fixtures if it's in the same directory as the test modules. If it's outside the directory, Pytest won't recognize it.