**PYTEST FRAMEWORK**

* widely used testing framework for Python.
* It provides a simple and flexible way to write tests for your Python code.
* Pytest supports a variety of testing types, including unit tests, functional tests, and integration tests.
* With Pytest, you can define your tests as regular Python functions, using assertions to check the expected outcomes.
* It offers powerful features such as test discovery, fixtures, parameterized testing, test skipping and marking, and test coverage reporting.

**Install pytest:**

Step1: In Command prompt run …**pip install –U pytest**

Step2: Check version ….**pytest –version**

**Note**: if version not show but display path error then we have to define the path on environmental variable seeting—system variable—path—edit—display path on command prompt copy this and paste it here—by new path click

Step3: check install pytest or not …**pip list**

Here show the list of application or liabrary show

Uninstall pytest ……**pip uninstall pytest**

**Write the test in pytest:**

1. Pytest file should have the format like **test\_name.py or name\_test.py**  eg test\_login.py
2. Test methods/ functions should **start with keyword “test”**

eg def testlogin():

1. **Note**: if you run test which are written in pycharm then its run succefully but don’t show any output. For that we have to run this program with pytest framework
2. **Run test on pytest framework:**

Go to run/ debug configuration window---(+)—pytest—select script path –apply ok—then run the test we will get the output

**Execute pytest on command prompt:**

1. Copy path of executable test from ----test—right click—copy path—absolute path
2. Command prompt:

**--- Cd paste path here + enter**

----**pytest + enter**

---**pytest -v (verbos)** ………..give more information about test

---**pytest -v –s /-vs** ….print statement/expresion also with information

--**pytest specific test name –vs** ………run specific test

-- **pytest specific test name –vsk log** ….run specific test which are having test name content log those test run only

**Grouping the test: mark decorator**

* is a powerful feature in Pytest that allows you to assign custom markers or attributes to your test functions.
* Markers provide a way to categorize, filter, and selectively run specific tests or groups of tests based on their assigned markers.

1. **Custome markers:**

Given markers name according to what we want

import pytest

def testlogin():  
 print('login succefully')  
  
def testlogoff():  
 print('logoff successful')

@pytest.mark.sanity ……**markers name sanity**

def testcalculation(): …this test case only run  
 assert 2+2==4

1. **Built in markers:**

* built-in markers that you can use to annotate your test functions and control their behavior.
* These markers allow you to skip tests, parameterize them, mark expected failures, and more. Here are some commonly used built-in markers in Pytest:

1. **@pytest.mark.skip**:

* This marker allows you to skip a test.
* When applied to a test function, the test will be marked as skipped and will not be executed.
* You can provide a reason for skipping as an optional argument

import pytest

@pytest.mark.skip(reason="Test currently under development")

def test\_function():

# Test implementation

1. **@pytest.mark.parametrize**:

* This marker allows you to define multiple sets of inputs for a single test function.
* It takes one or more arguments representing the parameter names and a list of values for each parameter.
* The test function will be executed multiple times, once for each combination of input values.

import pytest

@pytest.mark.parametrize("x, y, expected", [(2, 3, 5), (4, 6, 10)])

def test\_addition(x, y, expected):

assert x + y == expected

1. **@pytest.mark.xfail**:

* This marker marks a test as an expected failure.
* The test is executed, and if it fails, it is reported as an expected failure rather than an actual test failure.
* This is useful when you have tests that are known to fail due to a bug or an incomplete

import pytest

@pytest.mark.xfail

def test\_function():

# Test implementation that is expected to fail

1. **@pytest.mark.parametrize** and **@pytest.mark.xfail** can be combined to mark parameterized tests as expected failures. In this case, if any of the parameterized instances fail, they will be reported as expected failures rather than test failures.
2. **@pytest.mark.filterwarnings(warning**): add a warning filter to the given test.
3. **@pytest.mark.skipif(condition, ..., \*, reason=...)**: skip the given test function if any of the conditions evaluate to True.

Example: skipif(sys.platform == 'win32') skips the test if we are on the win32 platform.

1. **@pytest.mark.usefixtures(fixturename1, fixturename2, ...)**: mark tests as needing all of the specified fixtures.
2. **@pytest.mark.tryfirst**: mark a hook implementation function such that the plugin machinery will try to call it first/as early as possible. DEPRECATED, use @pytest.hookimpl(tryfirst=True) instead.
3. **@pytest.mark.trylast**: mark a hook implementation function such that the plugin machinery will try to call it last/as late as possible. DEPRECATED, use @pytest.hookimpl(trylast=True) instead.

**Fixture in pytest:**

* Fixtures are an essential feature in Pytest that provide a way to manage and set up the necessary resources or test dependencies for your tests.
* Fixtures help you write clean and reusable test code by separating the test setup from the test logic.
* A fixture in Pytest is a Python function that is defined with the **@pytest.fixture** decorator. It can be used in test functions by including it as an argument.
* Pytest will automatically invoke the fixture function and provide its returned value as the argument to the test function.
* Fixtures can be used in multiple test functions by simply including them as arguments in those functions.
* The **yield** statement acts as a separation point between the setup and teardown code. After the test completes, the teardown code is executed to clean up the resources.

import pytest

@pytest.fixture()  
def **setup**():  
 print('launch browser')  
 print('login')  
 print('browse products')  
 yield  
 *# print('logoff')  
 # print('close browser')*def testlogin(**setup**):  
 print('login succefully')  
  
def testlogoff():  
 print('logoff successful')

output:

test\_First.py::testlogin launch browser

login

browse products

login succefully

PASSED

test\_First.py::testlogoff logoff successful

PASSEd

**How to use conftest:**

* **conftest.py** file is a special file used in Pytest to define fixtures, plugins, and other configuration options that are shared across multiple test files or directories.
* It serves as a central place to configure and customize the behavior of your tests.
* When Pytest runs, it automatically discovers and applies the configuration defined in the **conftest.py** file located in the current directory or any of its ancestor directories.
* This allows you to define fixtures and other configurations that are accessible to all tests within that directory and its subdirectories.
* Conftest file content all the fixture those fixture we no need to write again and again in each test case . instead of this we just import pytest and call respective fixture method name as argument in the different function or method in the executable test case.
* If we use (**autouse=True** ) condtion in the fixture while defining then we no need to use fixture method name as argument in the test case method. Test case executed all with fixture define in the conftest.

Conftest file content :

import pytest  
@pytest.fixture() …. **autocase=True**def **tc\_setup**():  
 print('launch browser')  
 print('login')  
 print('browse products')  
 yield  
 print('logoff')  
 print('close browser')

import pytest  
  
def testlogin():  
 print('login succefully')  
  
def testlogoff():  
 print('logoff successful')  
  
  
def testcalculation(**tc\_setup**): ….fixture call using method name in conftet  
 assert 2+2==4

output:

test\_First.py::testlogin login succefully

PASSED

test\_First.py::testlogoff logoff successful

PASSED

test\_First.py::testcalculation launch browser

login

browse products

PASSEDlogoff

close browser

**scope in the fixture:**

* **scope** parameter in Pytest fixtures allows you to control the lifetime and scope of the fixture instances.
* The scope determines how many times a fixture is set up and torn down during the execution of your tests and how widely it is shared among the tests.

1. **Function: default**

* The fixture is instantiated once for each test function that uses it.
* The fixture is set up before the test function runs and torn down after the test function completes.

1. **Class:**

* The fixture is instantiated once for each test class that uses it.
* The fixture is set up before any test methods in the class are run and torn down after all test methods in the class have completed.

1. **Module:**

* The fixture is instantiated once for each test module that uses it.
* The fixture is set up before any tests in the module are run and torn down after all tests in the module have completed.

1. **Session:**

* the fixture is instantiated once for the entire duration of the test session.
* The fixture is set up before any tests start running and torn down after all tests have completed.