PyTest

Testing FrameWork

List Of Python Testing Frameworks

- Robot.
- PyTest.
- Unittest.
- DocTest.
- Nose2.
- Testify.

Advantages of Pytest

The advantages of Pytest are as follows −

- **Pytest can run multiple tests in parallel, which reduces the execution time of the test suite.**
- **Pytest has its own way to detect the test file and test functions automatically, if not mentioned explicitly.**
- **Pytest allows us to skip a subset of the tests during execution.**
- **Pytest allows us to run a subset of the entire test suite.**
- **Pytest is free and open source.**
- **Because of its simple syntax, pytest is very easy to start with.**

Installation pytest module

```
pip install pytest
```

Running pytest without mentioning a filename will run all files of format **test_*.py** or ***_test.py** in the current directory and subdirectories. Pytest automatically identifies those files as test files. We can make pytest run other filenames by explicitly mentioning them.

Pytest requires the test function names to start with test. Function names which are not of format **test*** are not considered as test functions by pytest. We cannot explicitly make pytest consider any function not starting with test as a test function.

Now, we will start with our first pytest program. We will first create a directory and thereby, create our test files in the directory.

Let us follow the steps shown below −

- Create a new directory named automation and navigate into the directory in your command line.
- Create a file named test_square.py and add the below code to that file.

```
import math

def test_sqrt():
   num = 25
   assert math.sqrt(num) == 5

def testsquare():
   num = 7
   assert 7*7 == 40

def tesequality():
   assert 10 == 11
```

Run the test using the following command −

```
pytest
```

The above command will generate the following output −

```
test_square.py .F
============================================ FAILURES
============================================
_____________________________________________ testsquare
_____________________________________________
   def testsquare():
   num=7
>  assert 7*7 == 40
E  assert (7 * 7) == 40
test_square.py:9: AssertionError
================================ 1 failed, 1 passed in 0.06 seconds
================================
```

we will learn how to execute single test file and multiple test files. We already have a test file test_square.py created. Create a new test file test_compare.py with the following code −

```
def test_greater():
   num = 100
   assert num > 100

def test_greater_equal():
   num = 100
   assert num >= 100

def test_less():
   num = 100
   assert num < 200
```

Run specific file name, -v is a flag. Gives details of Test cases in it's format.
-rA flag also to get details.

```
pytest test_compare.py -v
```

```
test_compare.py::test_greater FAILED
test_compare.py::test_greater_equal PASSED
test_compare.py::test_less PASSED
test_square.py::test_sqrt PASSED
test_square.py::testsquare FAILED
============================================== FAILURES
==============================================
_____________________________________________ test_greater
_____________________________________________
   def test_greater():
   num = 100
>  assert num > 100
E  assert 100 > 100

test_compare.py:3: AssertionError
_____________________________________________ testsquare
_____________________________________________
   def testsquare():
   num = 7
>  assert 7*7 == 40
E  assert (7 * 7) == 40

test_square.py:9: AssertionError
================================== 2 failed, 3 passed in 0.07 seconds
==================================
```

Now, run the following command −

```
pytest -k great -v
```

This will execute all the test names having the word **‘great’** in its name. In this case, they are **test_greater()** and **test_greater_equal()**. See the result below.

```
test_compare.py::test_greater FAILED
test_compare.py::test_greater_equal PASSED
============================================= FAILURES
=============================================
_____________________________________________ test_greater
_____________________________________________
def test_greater():
num = 100
>  assert num > 100
E  assert 100 > 100
test_compare.py:3: AssertionError
========================== 1 failed, 1 passed, 3 deselected in 0.07 seconds
==========================
```

Pytest allows us to use markers on test functions.

Markers are used to set various features/attributes to test functions. Pytest provides many inbuilt markers such as xfail, skip and parametrize. Apart from that, users can create their own marker names. Markers are applied on the tests using the syntax given below −

```
@pytest.mark.<markername>
```

`pytest` provides a few marks out of the box:

- **`skip` skips a test unconditionally.**
- **`skipif` skips a test if the expression passed to it evaluates to `True`.**
- **`xfail` indicates that a test is expected to fail, so if the test *does* fail, the overall suite can still result in a passing status.**
- **`parametrize` (note the spelling) creates multiple variants of a test with different values as arguments. You'll learn more about this mark shortly.**

Python

```python
@pytest.mark.parametrize("palindrome", [
    "",
    "a",
    "Bob",
    "Never odd or even",
    "Do geese see God?",
])
def test_is_palindrome(palindrome):
    assert is_palindrome(palindrome)

@pytest.mark.parametrize("non_palindrome", [
    "abc",
    "abab",
])
def test_is_palindrome_not_palindrome(non_palindrome):
    assert not is_palindrome(non_palindrome)
```

The first argument to `parametrize()` is a comma-delimited string of parameter names. The second argument is a list of either tuples or single values that represent the parameter value(s). You could take your parametrization a step further to combine all your tests into one:

Python

```python
@pytest.mark.parametrize("maybe_palindrome, expected_result", [
    ("", True),
    ("a", True),
    ("Bob", True),
    ("Never odd or even", True),
    ("Do geese see God?", True),
    ("abc", False),
    ("abab", False),
])
def test_is_palindrome(maybe_palindrome, expected_result):
    assert is_palindrome(maybe_palindrome) == expected_result
```

Parameterizing of a test is done to run the test against multiple sets of inputs. We can do this by using the following marker −

```
@pytest.mark.parametrize
```

Copy the below code into a file called **test_multiplication.py** −

```
import pytest

@pytest.mark.parametrize("num, output",[(1,11),(2,22),(3,35),(4,44)])
def test_multiplication_11(num, output):
   assert 11*num == output
```

Here the test multiplies an input with 11 and compares the result with the expected output. The test has 4 sets of inputs, each has 2 values – one is the number to be multiplied with 11 and the other is the expected result.

Execute the test by running the following command −

```
Pytest -k multiplication -v
```

```
test_multiplication.py::test_multiplication_11[1-11] PASSED
test_multiplication.py::test_multiplication_11[2-22] PASSED
test_multiplication.py::test_multiplication_11[3-35] FAILED
test_multiplication.py::test_multiplication_11[4-44] PASSED
============================================== FAILURES
==============================================
_________________ test_multiplication_11[3-35] _________________
num = 3, output = 35
   @pytest.mark.parametrize("num, output",[(1,11),(2,22),(3,35),(4,44)])
   def test_multiplication_11(num, output):
>  assert 11*num == output
E  assert (11 * 3) == 35
test_multiplication.py:5: AssertionError
============================== 1 failed, 3 passed, 8 deselected in 0.08 seconds
==============================
```