Pytest - Quick Guide

Pytest - Introduction

Pytest is a python based testing framework, which is used to write and execute test codes. In the present days of REST services, pytest is mainly used for API testing even though we can use pytest to write simple to complex tests, i.e., we can write codes to test API, database, UI, etc.

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.

In this tutorial, we will explain the pytest fundamentals with sample programs.

Pytest - Environment Setup

In this chapter, we will learn how to install pytest.

To start the installation, execute the following command –

```
pip install pytest == 2.9.1
```

We can install any version of pytest. Here, 2.9.1 is the version we are installing.

To install the latest version of pytest, execute the following command –

```
pip install pytest
```

Confirm the installation using the following command to display the help section of pytest.

```
pytest -h
```

Identifying Test files and Test Functions

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.

We will understand the execution of tests in our subsequent chapters.

Pytest - Starting With Basic Test

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
==================================
```

See the first line of the result. It displays the file name and the results. F represents a test failure and dot(.) represents a test success.

Below that, we can see the details of the failed tests. It will show at which statement the test has failed. In our example, 7*7 is compared for equality against 49, which is wrong. In the end, we can see test execution summary, 1 failed and 1 passed.

The function tescompare is not executed because pytest will not consider it as a test since its name is not of the format **test***.

Now, execute the below command and see the result again −

```
pytest -v
```

-v increases the verbosity.

```
test_square.py::test_sqrt PASSED
test_square.py::testsquare FAILED
========================================== 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 0.04 seconds
==================================
```

Now the result is more explanatory about the test that failed and the test that passed.

Note – pytest command will execute all the files of format **test_*** or ***_test** in the current directory and subdirectories.

Pytest - File Execution

In this chapter, 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
```

Now to run all the tests from all the files (2 files here) we need to run the following command –

```
pytest -v
```

The above command will run tests from both **test_square.py** and **test_compare.py**. The output will be generated as follows –

```
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 secon
====================================
```

To execute the tests from a specific file, use the following syntax −

```
pytest <filename> -v
```

Now, run the following command −

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

The above command will execute the tests only from file **test_compare.py.** Our result will be −

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

```
E  assert 100 > 100
test_compare.py:3: AssertionError
================================== 1 failed, 2 passed in 0.04 seconds
==================================
```

Execute a Subset of Test Suite

In a real scenario, we will have multiple test files and each file will have a number of tests. Tests will cover various modules and functionalities. Suppose, we want to run only a specific set of tests; how do we go about it?

Pytest provides two ways to run the subset of the test suite.

- Select tests to run based on substring matching of test names.
- Select tests groups to run based on the markers applied.

We will explain these two with examples in subsequent chapters.

Substring Matching of Test Names

To execute the tests containing a string in its name we can use the following syntax –

```
pytest -k <substring> -v
```

-k <substring> represents the substring to search for in the test names.

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 second
==========================
```

Here in the result, we can see 3 tests deselected. This is because those test names do not contain the word **great** in them.

Note – The name of the test function should still start with 'test'.

Pytest - Grouping the Tests

In this chapter, we will learn how to group the tests using markers.

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>
```

To use markers, we have to **import pytest** module in the test file. We can define our own marker names to the tests and run the tests having those marker names.

To run the marked tests, we can use the following syntax –

```
pytest -m <markername> -v
```

-m <markername> represents the marker name of the tests to be executed.

Update our test files **test_compare.py** and **test_square.py** with the following code. We are defining 3 markers **– great, square, others**.

test_compare.py

```
import pytest
@pytest.mark.great
def test_greater():
   num = 100
   assert num > 100
```

```
@pytest.mark.great
def test_greater_equal():
   num = 100
   assert num >= 100

@pytest.mark.others
def test_less():
   num = 100
   assert num < 200
```

Explore our **latest online courses** and learn new skills at your own pace. Enroll and become a certified expert to boost your career.

test_square.py

```
import pytest
import math

@pytest.mark.square
def test_sqrt():
   num = 25
   assert math.sqrt(num) == 5

@pytest.mark.square
def testsquare():
   num = 7
   assert 7*7 == 40

@pytest.mark.others
   def test_equality():
   assert 10 == 11
```

Now to run the tests marked as **others**, run the following command −

```
pytest -m others -v
```

See the result below. It ran the 2 tests marked as **others**.

```
test_compare.py::test_less PASSED
test_square.py::test_equality FAILED
================================================ FAILURES
================================================
```

```
_____________________________________________ test_equality
_____________________________________________
   @pytest.mark.others
   def test_equality():
>  assert 10 == 11
E  assert 10 == 11
test_square.py:16: AssertionError
========================== 1 failed, 1 passed, 4 deselected in 0.08 second
==========================
```

Similarly, we can run tests with other markers also – great, compare

Pytest - Fixtures

Fixtures are functions, which will run before each test function to which it is applied. Fixtures are used to feed some data to the tests such as database connections, URLs to test and some sort of input data. Therefore, instead of running the same code for every test, we can attach fixture function to the tests and it will run and return the data to the test before executing each test.

A function is marked as a fixture by –

```
@pytest.fixture
```

A test function can use a fixture by mentioning the fixture name as an input parameter.

Create a file **test_div_by_3_6.py** and add the below code to it

```
import pytest

@pytest.fixture
def input_value():
   input = 39
   return input

def test_divisible_by_3(input_value):
   assert input_value % 3 == 0

def test_divisible_by_6(input_value):
   assert input_value % 6 == 0
```

Here, we have a fixture function named **input_value**, which supplies the input to the tests. To access the fixture function, the tests have to mention the fixture name as input

parameter.

Pytest while the test is getting executed, will see the fixture name as input parameter. It then executes the fixture function and the returned value is stored to the input parameter, which can be used by the test.

Execute the test using the following command –

```
pytest -k divisible -v
```

The above command will generate the following result –

```
test_div_by_3_6.py::test_divisible_by_3 PASSED
test_div_by_3_6.py::test_divisible_by_6 FAILED
================================================ FAILURES
================================================
_____________________________________________ test_divisible_by_6
_____________________________________________
input_value = 39
   def test_divisible_by_6(input_value):
>  assert input_value % 6 == 0
E  assert (39 % 6) == 0
test_div_by_3_6.py:12: AssertionError
========================== 1 failed, 1 passed, 6 deselected in 0.07 second
==========================
```

However, the approach comes with its own limitation. A fixture function defined inside atest file has a scope within the test file only. We cannot use that fixture in another test file. To make a fixture available to multiple test files, we have to define the fixture function in a file called conftest.py. **conftest.py** is explained in the next chapter.

Pytest - Conftest.py

We can define the fixture functions in this file to make them accessible across multiple test files.

Create a new file **conftest.py** and add the below code into it –

```
import pytest

@pytest.fixture
def input_value():
```

```
    input = 39
    return input
```

Edit the **test_div_by_3_6.py** to remove the fixture function −

```
import pytest

def test_divisible_by_3(input_value):
   assert input_value % 3 == 0

def test_divisible_by_6(input_value):
   assert input_value % 6 == 0
```

Create a new file **test_div_by_13.py** −

```
import pytest

def test_divisible_by_13(input_value):
   assert input_value % 13 == 0
```

Now, we have the files **test_div_by_3_6.py** and **test_div_by_13.py** making use of the fixture defined in **conftest.py**.

Run the tests by executing the following command −

```
pytest -k divisible -v
```

The above command will generate the following result −

```
test_div_by_13.py::test_divisible_by_13 PASSED
test_div_by_3_6.py::test_divisible_by_3 PASSED
test_div_by_3_6.py::test_divisible_by_6 FAILED
================================================ FAILURES
================================================
_____________________________________________ test_divisible_by_6
_____________________________________________
input_value = 39
   def test_divisible_by_6(input_value):
>  assert input_value % 6 == 0
E  assert (39 % 6) == 0
test_div_by_3_6.py:7: AssertionError
```

```
========================== 1 failed, 2 passed, 6 deselected in 0.09 second
==========================
```

The tests will look for fixture in the same file. As the fixture is not found in the file, it will check for fixture in conftest.py file. On finding it, the fixture method is invoked and the result is returned to the input argument of the test.

Pytest - Parameterizing Tests

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
```

The above command will generate the following output –

```
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
================================
```

Pytest - Xfail/Skip Tests

In this chapter, we will learn about the Skip and Xfail tests in Pytest.

Now, consider the below situations –

- A test is not relevant for some time due to some reasons.
- A new feature is being implemented and we already added a test for that feature.

In these situations, we have the option to xfail the test or skip the tests.

Pytest will execute the xfailed test, but it will not be considered as part failed or passed tests. Details of these tests will not be printed even if the test fails (remember pytest usually prints the failed test details). We can xfail tests using the following marker –

```
@pytest.mark.xfail
```

Skipping a test means that the test will not be executed. We can skip tests using the following marker –

```
@pytest.mark.skip
```

Later, when the test becomes relevant we can remove the markers.

Edit the **test_compare.py** we already have to include the xfail and skip markers –

```
import pytest
@pytest.mark.xfail
@pytest.mark.great
def test_greater():
   num = 100
   assert num > 100

@pytest.mark.xfail
```

```
@pytest.mark.great
def test_greater_equal():
   num = 100
   assert num >= 100

@pytest.mark.skip
@pytest.mark.others
def test_less():
   num = 100
   assert num < 200
```

Execute the test using the following command –

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

Upon execution, the above command will generate the following result –

```
test_compare.py::test_greater xfail
test_compare.py::test_greater_equal XPASS
test_compare.py::test_less SKIPPED
============================== 1 skipped, 1 xfailed, 1 xpassed in 0.06 secc
============================
```

Pytest - Stop Test Suite after N Test Failures

In a real scenario, once a new version of the code is ready to deploy, it is first deployed into pre-prod/ staging environment. Then a test suite runs on it.

The code is qualified for deploying to production only if the test suite passes. If there is test failure, whether it is one or many, the code is not production ready.

Therefore, what if we want to stop the execution of test suite soon after n number of test fails. This can be done in pytest using maxfail.

The syntax to stop the execution of test suite soon after n number of test fails is as follows –

```
pytest --maxfail = <num>
```

Create a file test_failure.py with the following code.

```
import pytest
import math

def test_sqrt_failure():
   num = 25
   assert math.sqrt(num) == 6

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

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

All the 3 tests will fail on executing this test file. Here, we are going to stop the execution of the test after one failure itself by −

```
pytest test_failure.py -v --maxfail 1
```

```
test_failure.py::test_sqrt_failure FAILED
==================================== FAILURES
==================================== _____________________________
test_sqrt_failure ____________________________________________
  def test_sqrt_failure():
  num = 25
>  assert math.sqrt(num) == 6
E  assert 5.0 == 6
E  + where 5.0 = <built-in function sqrt>(25)
E  + where <built-in function sqrt>= math.sqrt
test_failure.py:6: AssertionError
================================== 1 failed in 0.04 seconds
==================================
```

In the above result, we can see the execution is stopped on one failure.

Pytest - Run Tests in Parallel

By default, pytest runs tests in sequential order. In a real scenario, a test suite will have a number of test files and each file will have a bunch of tests. This will lead to a large execution time. To overcome this, pytest provides us with an option to run tests in parallel.

For this, we need to first install the pytest-xdist plugin.

Install pytest-xdist by running the following command −

```
pip install pytest-xdist
```

Now, we can run tests by using the syntax **pytest -n <num>**

```
pytest -n 3
```

-n <num> runs the tests by using multiple workers, here it is 3.

We will not be having much time difference when there is only a few tests to run. However, it matters when the test suite is large.

Test Execution Results in XML Format

We can generate the details of the test execution in an xml file. This xml file is mainly useful in cases where we have a dashboard that projects the test results. In such cases, the xml can be parsed to get the details of the execution.

We will now execute the tests from test_multiplcation.py and generate the xml by running

```
pytest test_multiplication.py -v --junitxml="result.xml"
```

Now we can see result.xml is generated with the following data −

```
<?xml version = "1.0" encoding = "utf-8"?>
<testsuite errors = "0" failures = "1"
name = "pytest" skips = "0" tests = "4" time = "0.061">
   <testcase classname = "test_multiplication"
      file = "test_multiplication.py"
      line = "2" name = "test_multiplication_11[1-11]"
      time = "0.00117516517639>
   </testcase>

   <testcase classname = "test_multiplication"
      file = "test_multiplication.py"
      line = "2" name = "test_multiplication_11[2-22]"
      time = "0.00155973434448">
   </testcase>
```

```
  <testcase classname = "test_multiplication"
    file = "test_multiplication.py"
    line = "2" name = "test_multiplication_11[3-35]" time = "0.00144290924072">
    failure message = "assert (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: AssertionErro
    </failure>
  </testcase>
  <testcase classname = "test_multiplication"
    file = "test_multiplication.py"
    line = "2" name = "test_multiplication_11[4-44]"
    time = "0.000945091247559">
  </testcase>
</testsuite>
```

Here, the tag **<testsuit>** summarises there were 4 tests and the number of failures are 1.

- The tag **<testcase>** gives the details of each executed test.
- <failure> tag gives the details of the failed test code.

Pytest - Summary

In this pytest tutorial, we covered the following areas –

- Installing pytest..
- Identifying test files and test functions.
- Executing all test files using pytest –v.
- Executing specific file usimng pytest <filename> -v.
- Execute tests by substring matching pytest -k <substring> -v.
- Execute tests based on markers pytest -m <marker_name> -v.
- Creating fixtures using @pytest.fixture.

- conftest.py allows accessing fixtures from multiple files.
- Parametrizing tests using @pytest.mark.parametrize.
- Xfailing tests using @pytest.mark.xfail.
- Skipping tests using @pytest.mark.skip.
- Stop test execution on n failures using pytest --maxfail = <num>.
- Running tests in parallel using pytest -n <num>.
- Generating results xml using pytest -v --junitxml = "result.xml".

Pytest - Conclusion

This tutorial introduced you to pytest framework. Now you should be able to start writing tests using pytest.

As a good practice −

- Create different test files based on functionality/module being tested.
- Give meaningful names to test files and methods.
- Have enough markers to group the tests based on various criteria.
- Use fixtures whenever needed.