**PyTest Tutorial: What is, Install, Fixture, Assertions**

**What is PyTest?**

Pytest is a testing framework which allows us to write test codes using python. You can write code to test anything like database , API, even UI if you want. But pytest is mainly being used in industry to write tests for APIs.

**Why use PyTest?**

Some of the advantages of pytest are

* Very easy to start with because of its simple and easy syntax.
* Can run tests in parallel.
* Can run a specific test or a subset of tests
* Automatically detect tests
* Skip tests
* Open source

In this tutorial, you will learn:

* [What is PyTest?](https://www.guru99.com/pytest-tutorial.html#1)
* [Why use PyTest?](https://www.guru99.com/pytest-tutorial.html#2)
* [How to install PyTest](https://www.guru99.com/pytest-tutorial.html#3)
* [First Basic PyTest](https://www.guru99.com/pytest-tutorial.html#4)
* [Assertions in PyTest](https://www.guru99.com/pytest-tutorial.html#5)
* [How pytest identifies the test files and test methods](https://www.guru99.com/pytest-tutorial.html#6)
* [Run multiple tests from a specific file and multiple files.](https://www.guru99.com/pytest-tutorial.html#7)
* [Run a subset of entire test](https://www.guru99.com/pytest-tutorial.html#8)
* [Running tests in parallel](https://www.guru99.com/pytest-tutorial.html#9)
* [Pytest fixtures](https://www.guru99.com/pytest-tutorial.html#10)
* [Parameterized tests](https://www.guru99.com/pytest-tutorial.html#11)
* [Xfail / Skip tests](https://www.guru99.com/pytest-tutorial.html#12)
* [Results XML](https://www.guru99.com/pytest-tutorial.html#13)
* [A pytest framework testing an API](https://www.guru99.com/pytest-tutorial.html#14)

**How to install PyTest**

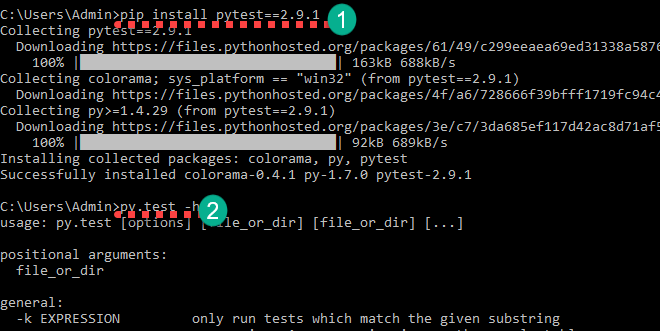
**Step 1)** You can install pytest by

pip install pytest==2.9.1

Once the installation is complete you can confirm it with by

py.test -h

This will display the help

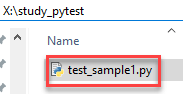
[](https://www.guru99.com/images/1/011019_1320_PyTestTutor1.png)

**First Basic PyTest**

Create a folder study\_pytest. We are going to create our test files inside this folder.

Please navigate to that folder in your command line.

Create a file named test\_sample1.py inside the folder

[](https://www.guru99.com/images/1/011019_1320_PyTestTutor2.png)

Add the below code into it and save

import pytest

def test\_file1\_method1():

x=5

y=6

assert x+1 == y,"test failed"

assert x == y,"test failed"

def test\_file1\_method2():

x=5

y=6

assert x+1 == y,"test failed"

Run the test using the command

py.test

You'll get output as

test\_sample1.py F.

============================================== FAILURES ========================================

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ test\_sample1 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

def test\_file1\_method1():

x=5

y=6

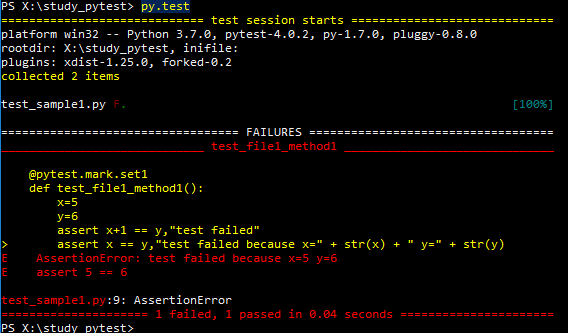
assert x+1 == y,"test failed"

> assert x == y,"test failed"

E AssertionError: test failed

E assert 5 == 6

test\_sample1.py:6: AssertionError

[](https://www.guru99.com/images/1/011019_1320_PyTestTutor3.png)

Here in test\_sample1.py F.

F says failure

Dot(.) says success.

In the failures section, you can see the failed method(s) and the line of failure. Here x==y means 5==6 which is false.

**Assertions in PyTest**

Assertions are checks that return either True or False status. In pytest, if an assertion fails in a test method, then that method execution is stopped there. The remaining code in that test method is not executed, and pytest will continue with the next test method.

**Examples:**

assert "hello" == "Hai" is an assertion failure.

assert 4==4 is a successful assertion

assert True is a successful assertion

assert False is an assertion failure.

Consider

assert x == y,"test failed because x=" + str(x) + " y=" + str(y)

Place this code in test\_file1\_method1() instead of the assertion

assert x == y,"test failed"

Running the test will give the failure as **AssertionError: test failed x=5 y=6**

**How pytest identifies the test files and test methods**

By default pytest only identifies the file names starting with **test\_** or ending with **\_test** as the test files. We can explicitly mention other filenames though (explained later). Pytest requires the test method names to start with **"test**." All other method names will be ignored even if we explicitly ask to run those methods.

See some examples of valid and invalid pytest file names

test\_login.py - valid

login\_test.py - valid

testlogin.py -invalid

logintest.py -invalid

Note: Yes we can explicitly ask pytest to pick testlogin.py and logintest.py

See some examples of valid and invalid pytest test methods

def test\_file1\_method1(): - valid

def testfile1\_method1(): - valid

def file1\_method1(): - invalid

Note: Even if we explicitly mention file1\_method1() pytest will not run this method.

**Run multiple tests from a specific file and multiple files.**

Currently, inside the folder study\_pytest, we have a file test\_sample1.py. Suppose we have multiple files , say test\_sample2.py , test\_sample3.py. To run all the tests from all the files in the folder and subfolders we need to just run the pytest command.

py.test

This will run all the filenames starting with test\_ and the filenames ending with \_test in that folder and subfolders under that folder.

To run tests only from a specific file, we can use py.test <filename>

py.test test\_sample1.py

**Run a subset of entire test**

Sometimes we don't want to run the entire test suite. Pytest allows us to run specific tests. We can do it in 2 ways

* Grouping of test names by substring matching
* Grouping of tests by markers

We already have test\_sample1.py. Create a file test\_sample2.py and add the below code into it

def test\_file2\_method1():

x=5

y=6

assert x+1 == y,"test failed"

assert x == y,"test failed because x=" + str(x) + " y=" + str(y)

def test\_file2\_method2():

x=5

y=6

assert x+1 == y,"test failed"

So we have currently

* test\_sample1.py
  + test\_file1\_method1()
  + test\_file1\_method2()
* test\_sample2.py
  + test\_file2\_method1()
  + test\_file2\_method2()

**Option 1) Run tests by substring matching**

Here to run all the tests having method1 in its name we have to run

py.test -k method1 -v

-k <expression> is used to represent the substring to match

-v increases the verbosity

So running py.test -k method1 -v will give you the following result

test\_sample2.py::test\_file2\_method1 FAILED

test\_sample1.py::test\_file1\_method1 FAILED

============================================== FAILURES ==============================================

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ test\_file2\_method1 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

def test\_file2\_method1():

x=5

y=6

assert x+1 == y,"test failed"

> assert x == y,"test failed because x=" + str(x) + " y=" + str(y)

E AssertionError: test failed because x=5 y=6

E assert 5 == 6

test\_sample2.py:5: AssertionError

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ test\_file1\_method1 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

@pytest.mark.only

def test\_file1\_method1():

x=5

y=6

assert x+1 == y,"test failed"

> assert x == y,"test failed because x=" + str(x) + " y=" + str(y)

E AssertionError: test failed because x=5 y=6

E assert 5 == 6

test\_sample1.py:8: AssertionError

================================= 2 tests deselected by '-kmethod1' ==================================

=============================== 2 failed, 2 deselected in 0.02 seconds ===============================

Here you can see towards the end **2 tests deselected by '-kmethod1'**which are test\_file1\_method2 and test\_file2\_method2

Try running with various combinations like:-

py.test -k method -v - will run all the four methods

py.test -k methods -v – will not run any test as there is no test name matches the substring 'methods'

**Option 2) Run tests by markers**

Pytest allows us to set various attributes for the test methods using pytest markers, @pytest.mark . To use markers in the test file, we need to import pytest on the test files.

Here we will apply different marker names to test methods and run specific tests based on marker names. We can define the markers on each test names by using

@pytest.mark.<name>.

We are defining markers set1 and set2 on the test methods, and we will run the test using the marker names. Update the test files with the following code

test\_sample1.py

import pytest

@pytest.mark.set1

def test\_file1\_method1():

x=5

y=6

assert x+1 == y,"test failed"

assert x == y,"test failed because x=" + str(x) + " y=" + str(y)

@pytest.mark.set2

def test\_file1\_method2():

x=5

y=6

assert x+1 == y,"test failed"

test\_sample2.py

import pytest

@pytest.mark.set1

def test\_file2\_method1():

x=5

y=6

assert x+1 == y,"test failed"

assert x == y,"test failed because x=" + str(x) + " y=" + str(y)

@pytest.mark.set1

def test\_file2\_method2():

x=5

y=6

assert x+1 == y,"test failed"

We can run the marked test by

py.test -m <name>

-m <name> mentions the marker name

Run py.test -m set1.This will run the methods test\_file1\_method1, test\_file2\_method1, test\_file2\_method2.

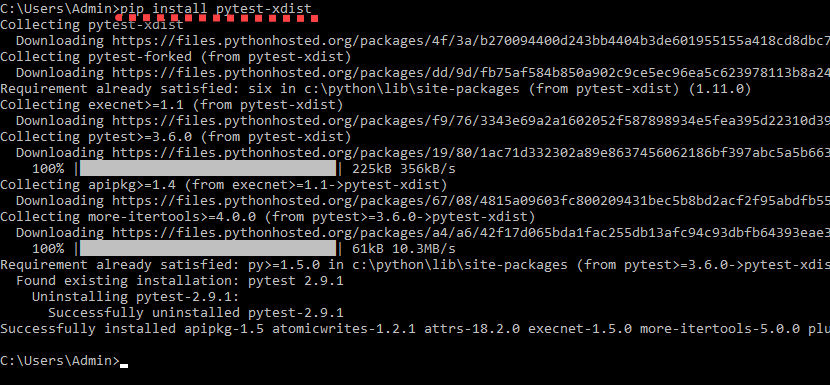
Running py.test -m set2 will run test\_file1\_method2.

**Running tests in parallel**

Usually, a test suite will have multiple test files and hundreds of test methods which will take a considerable amount of time to execute. Pytest allows us to run tests in parallel.

For that we need to first install pytest-xdist by running

pip install pytest-xdist

[](https://www.guru99.com/images/1/011019_1320_PyTestTutor4.png)

You can run tests now by

py.test -n 4

-n <num> runs the tests by using multiple workers. In the above command, there will be 4 workers to run the test.

**Pytest fixtures**

Fixtures are used when we want to run some code before every test method. So instead of repeating the same code in every test we define fixtures. Usually, fixtures are used to initialize database connections, pass the base , etc

A method is marked as a fixture by marking with

@pytest.fixture

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

Create a new file test\_basic\_fixture.py with following code

import pytest

@pytest.fixture

def supply\_AA\_BB\_CC():

aa=25

bb =35

cc=45

return [aa,bb,cc]

def test\_comparewithAA(supply\_AA\_BB\_CC):

zz=35

assert supply\_AA\_BB\_CC[0]==zz,"aa and zz comparison failed"

def test\_comparewithBB(supply\_AA\_BB\_CC):

zz=35

assert supply\_AA\_BB\_CC[1]==zz,"bb and zz comparison failed"

def test\_comparewithCC(supply\_AA\_BB\_CC):

zz=35

assert supply\_AA\_BB\_CC[2]==zz,"cc and zz comparison failed"

Here

* We have a fixture named supply\_AA\_BB\_CC. This method will return a list of 3 values.
* We have 3 test methods comparing against each of the values.

Each of the test function has an input argument whose name is matching with an available fixture. Pytest then invokes the corresponding fixture method and the returned values will be stored in the input argument , here the list [25,35,45]. Now the list items are being used in test methods for the comparison.

Now run the test and see the result

py.test test\_basic\_fixture

test\_basic\_fixture.py::test\_comparewithAA FAILED

test\_basic\_fixture.py::test\_comparewithBB PASSED

test\_basic\_fixture.py::test\_comparewithCC FAILED

============================================== FAILURES ==============================================

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ test\_comparewithAA \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

supply\_AA\_BB\_CC = [25, 35, 45]

def test\_comparewithAA(supply\_AA\_BB\_CC):

zz=35

> assert supply\_AA\_BB\_CC[0]==zz,"aa and zz comparison failed"

E AssertionError: aa and zz comparison failed

E assert 25 == 35

test\_basic\_fixture.py:10: AssertionError

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ test\_comparewithCC \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

supply\_AA\_BB\_CC = [25, 35, 45]

def test\_comparewithCC(supply\_AA\_BB\_CC):

zz=35

> assert supply\_AA\_BB\_CC[2]==zz,"cc and zz comparison failed"

E AssertionError: cc and zz comparison failed

E assert 45 == 35

test\_basic\_fixture.py:16: AssertionError

================================= 2 failed, 1 passed in 0.05 seconds =================================

The test test\_comparewithBB is passed since zz=BB=35, and the remaining 2 tests are failed.

The fixture method has a scope only within that test file it is defined. If we try to access the fixture in some other test file , we will get an error saying fixture **'supply\_AA\_BB\_CC' not found**for the test methods in other files.

To use the same fixture against multiple test files, we will create fixture methods in a file called conftest.py.

Let's see this by the below example. Create 3 files conftest.py, test\_basic\_fixture.py, test\_basic\_fixture2.py with the following code

conftest.py

import pytest

@pytest.fixture

def supply\_AA\_BB\_CC():

aa=25

bb =35

cc=45

return [aa,bb,cc]

test\_basic\_fixture.py

import pytest

def test\_comparewithAA(supply\_AA\_BB\_CC):

zz=35

assert supply\_AA\_BB\_CC[0]==zz,"aa and zz comparison failed"

def test\_comparewithBB(supply\_AA\_BB\_CC):

zz=35

assert supply\_AA\_BB\_CC[1]==zz,"bb and zz comparison failed"

def test\_comparewithCC(supply\_AA\_BB\_CC):

zz=35

assert supply\_AA\_BB\_CC[2]==zz,"cc and zz comparison failed"

test\_basic\_fixture2.py

import pytest

def test\_comparewithAA\_file2(supply\_AA\_BB\_CC):

zz=25

assert supply\_AA\_BB\_CC[0]==zz,"aa and zz comparison failed"

def test\_comparewithBB\_file2(supply\_AA\_BB\_CC):

zz=25

assert supply\_AA\_BB\_CC[1]==zz,"bb and zz comparison failed"

def test\_comparewithCC\_file2(supply\_AA\_BB\_CC):

zz=25

assert supply\_AA\_BB\_CC[2]==zz,"cc and zz comparison failed"

pytest will look for the fixture in the test file first and if not found it will look in the conftest.py

Run the test by py.test -k test\_comparewith -v to get the result as below

test\_basic\_fixture.py::test\_comparewithAA FAILED

test\_basic\_fixture.py::test\_comparewithBB PASSED

test\_basic\_fixture.py::test\_comparewithCC FAILED

test\_basic\_fixture2.py::test\_comparewithAA\_file2 PASSED

test\_basic\_fixture2.py::test\_comparewithBB\_file2 FAILED

test\_basic\_fixture2.py::test\_comparewithCC\_file2 FAILED

**Parameterized tests**

The purpose of parameterizing a test is to run a test against multiple sets of arguments. We can do this by @pytest.mark.parametrize.

We will see this with the below example. Here we will pass 3 arguments to a test method. This test method will add the first 2 arguments and compare it with the 3rd argument.

Create the test file test\_addition.py with the below code

import pytest

@pytest.mark.parametrize("input1, input2, output",[(5,5,10),(3,5,12)])

def test\_add(input1, input2, output):

assert input1+input2 == output,"failed"

Here the test method accepts 3 arguments- input1, input2, output. It adds input1 and input2 and compares against the output.

Let's run the test by py.test -k test\_add -v and see the result

test\_addition.py::test\_add[5-5-10] PASSED

test\_addition.py::test\_add[3-5-12] FAILED

============================================== FAILURES ==============================================

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ test\_add[3-5-12] \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

input1 = 3, input2 = 5, output = 12

@pytest.mark.parametrize("input1, input2, output",[(5,5,10),(3,5,12)])

def test\_add(input1, input2, output):

> assert input1+input2 == output,"failed"

E AssertionError: failed

E assert (3 + 5) == 12

test\_addition.py:5: AssertionError

You can see the tests ran 2 times – one checking 5+5 ==10 and other checking 3+5 ==12

test\_addition.py::test\_add[5-5-10] PASSED

test\_addition.py::test\_add[3-5-12] FAILED

**Xfail / Skip tests**

There will be some situations where we don't want to execute a test, or a test case is not relevant for a particular time. In those situations, we have the option to xfail the test or skip the tests

The xfailed test will be executed, but it will not be counted as part failed or passed tests. There will be no traceback displayed if that test fails. We can xfail tests using

@pytest.mark.xfail.

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

@pytest.mark.skip.

Edit the test\_addition.py with the below code

import pytest

@pytest.mark.skip

def test\_add\_1():

assert 100+200 == 400,"failed"

@pytest.mark.skip

def test\_add\_2():

assert 100+200 == 300,"failed"

@pytest.mark.xfail

def test\_add\_3():

assert 15+13 == 28,"failed"

@pytest.mark.xfail

def test\_add\_4():

assert 15+13 == 100,"failed"

def test\_add\_5():

assert 3+2 == 5,"failed"

def test\_add\_6():

assert 3+2 == 6,"failed"

Here

* test\_add\_1 and test\_add\_2 are skipped and will not be executed.
* test\_add\_3 and test\_add\_4 are xfailed. These tests will be executed and will be part of xfailed(on test failure) or xpassed(on test pass) tests. There won't be any traceback for failures.
* test\_add\_5 and test\_add\_6 will be executed and test\_add\_6 will report failure with traceback while the test\_add\_5 passes

Execute the test by py.test test\_addition.py -v and see the result

test\_addition.py::test\_add\_1 SKIPPED

test\_addition.py::test\_add\_2 SKIPPED

test\_addition.py::test\_add\_3 XPASS

test\_addition.py::test\_add\_4 xfail

test\_addition.py::test\_add\_5 PASSED

test\_addition.py::test\_add\_6 FAILED

============================================== FAILURES ==============================================

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ test\_add\_6 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

def test\_add\_6():

> assert 3+2 == 6,"failed"

E AssertionError: failed

E assert (3 + 2) == 6

test\_addition.py:24: AssertionError

================ 1 failed, 1 passed, 2 skipped, 1 xfailed, 1 xpassed in 0.07 seconds =================

**Results XML**

We can create test results in XML format which we can feed to Continuous Integration servers for further processing and so. This can be done by

py.test test\_sample1.py -v --junitxml="result.xml"

The result.xml will record the test execution result. Find a sample result.xml below

<?xml version="1.0" encoding="UTF-8"?>

<testsuite errors="0" failures="1" name="pytest" skips="0" tests="2" time="0.046">

<testcase classname="test\_sample1" file="test\_sample1.py" line="3" name="test\_file1\_method1" time="0.001384973526">

<failure message="AssertionError:test failed because x=5 y=6 assert 5 ==6">

@pytest.mark.set1

def test\_file1\_method1():

x=5

y=6

assert x+1 == y,"test failed"

> assert x == y,"test failed because x=" + str(x) + " y=" + str(y)

E AssertionError: test failed because x=5 y=6

E assert 5 == 6

test\_sample1.py:9: AssertionError

</failure>

</testcase>

<testcase classname="test\_sample1" file="test\_sample1.py" line="10" name="test\_file1\_method2" time="0.000830173492432" />

</testsuite>

From <testsuite errors="0" failures="1" name="pytest" skips="0" tests="2" time="0.046"> we can see a total of two tests of which one is failed. Below that you can see the details regarding each executed test under <testcase> tag.

**A pytest framework testing an API**

Now we will create a small pytest framework to test an API. The API here used is a free one from <https://reqres.in/>. This website is just to provide testable API. This website doesn't store our data.

Here we will write some tests for

* listing some users
* login with users

Create the below files with the code given

conftest.py - have a fixture which will supply base url for all the test methods

import pytest

@pytest.fixture

def supply\_url():

return "https://reqres.in/api"

test\_list\_user.py – contains the test methods for listing valid and invalid users

* test\_list\_valid\_user tests for valid user fetch and verifies the response
* test\_list\_invaliduser tests for invalid user fetch and verifies the response

import pytest

import requests

import json

@pytest.mark.parametrize("userid, firstname",[(1,"George"),(2,"Janet")])

def test\_list\_valid\_user(supply\_url,userid,firstname):

url = supply\_url + "/users/" + str(userid)

resp = requests.get(url)

j = json.loads(resp.text)

assert resp.status\_code == 200, resp.text

assert j['data']['id'] == userid, resp.text

assert j['data']['first\_name'] == firstname, resp.text

def test\_list\_invaliduser(supply\_url):

url = supply\_url + "/users/50"

resp = requests.get(url)

assert resp.status\_code == 404, resp.text

test\_login\_user.py – contains test methods for testing login functionality.

* test\_login\_valid tests the valid login attempt with email and password
* test\_login\_no\_password tests the invalid login attempt without passing password
* test\_login\_no\_email tests the invalid login attempt without passing email.

import pytest

import requests

import json

def test\_login\_valid(supply\_url):

url = supply\_url + "/login/"

data = {'email':['test@test.com](mailto:'test@test.com)

','password':'something'}

resp = requests.post(url, data=data)

j = json.loads(resp.text)

assert resp.status\_code == 200, resp.text

assert j['token'] == "QpwL5tke4Pnpja7X", resp.text

def test\_login\_no\_password(supply\_url):

url = supply\_url + "/login/"

data = {'email':['test@test.com](mailto:'test@test.com)'}

resp = requests.post(url, data=data)

j = json.loads(resp.text)

assert resp.status\_code == 400, resp.text

assert j['error'] == "Missing password", resp.text

def test\_login\_no\_email(supply\_url):

url = supply\_url + "/login/"

data = {}

resp = requests.post(url, data=data)

j = json.loads(resp.text)

assert resp.status\_code == 400, resp.text

assert j['error'] == "Missing email or username", resp.text

Run the test using py.test -v

See the result as

test\_list\_user.py::test\_list\_valid\_user[1-George] PASSED

test\_list\_user.py::test\_list\_valid\_user[2-Janet] PASSED

test\_list\_user.py::test\_list\_invaliduser PASSED

test\_login\_user.py::test\_login\_valid PASSED

test\_login\_user.py::test\_login\_no\_password PASSED

test\_login\_user.py::test\_login\_no\_email PASSED

Update the tests and try various outputs

**Summary**

In this pytest tutorial, we covered

* Install pytest using pip install pytest=2.9.1
* Simple pytest program and run it with py.test command.
* Assertion statements, assert x==y, will return either True or False.
* How pytest identifies test files and methods.
  + Test files starting with **test\_** or ending with **\_test**
  + Test methods starting with **test**
* py.test command will run all the test files in that folder and subfolders. To run a specific file, we can use the command py.test <filename>
* Run a subset of test methods
  + Grouping of test names by substring matching.  
    py.test -k <name> -v will run all the tests having <name> in its name.
  + Run test by markers.Mark the tests using @pytest.mark.<name> and run the tests using pytest -m <name> to run tests marked as <name>.
* Run tests in parallel
  + Install pytest-xdist using pip install pytest-xdist
  + Run tests using py.test -n NUM where NUM is the number of workers
* Creating fixture methods to run code before every test by marking the method with @pytest.fixture
  + The scope of a fixture method is within the file it is defined.
  + A fixture method can be accessed across multiple test files by defining it in conftest.py file.
  + A test method can access a fixture by using it as an input argument.
* Parametrizing tests to run it against multiple set of inputs.  
  @pytest.mark.parametrize("input1, input2, output",[(5,5,10),(3,5,12)]) def test\_add(input1, input2, output):  
  assert input1+input2 == output,"failed"  
  will run the test with inputs (5,5,10) and (3,5,12)
* Skip/xfail tests using @pytets.mark.skip and @pytest.mark.xfail
* Create test results in XML format which covers executed test details using py.test test\_sample1.py -v --junitxml="result.xml"
* A sample pytest framework to test an API