**PYTEST:**

**Ref python programs: test\_fixures\_without.py,** **test\_with\_fixures.py,** **test\_closure. py, unit.py**

pytest is a software test framework, which means pytest is a command-line tool that automatically finds tests you’ve written, runs the tests, and reports the results. It has a library of goodies that you can use in your tests to help you test more effectively. It can be extended by writing plugins or installing third-party plugins. It can be used to test Python distributions. And it integrates easily with other tools like continuous integration and web automation.

Here are a few of the reasons pytest stands out above many other test frameworks:

* Simple tests are simple to write in pytest.
* Complex tests are still simple to write.
* Tests are easy to read. Tests are easy to read. (So important it’s listed twice.)
* You can get started in seconds.
* You use assert to fail a test, not things like self.assertEqual() or self.assertLessThan(). Just assert.
* You can use pytest to run tests written for unittest or nose.

Tests can be executed by using pytest in command prompt by opening the file path.

this file saved as sample.py in C:\Users\Admin\Desktop\sahitya\projects.

assert [1,2,3]=[1,2,3]

Lets run

cd ":\Users\Admin\Desktop\sahitya\projects"

pytest sample.py

============================= test session starts =============================

platform win32 -- Python 3.6.2, pytest-3.2.1, py-1.4.34, pluggy-0.4.0

rootdir: C:\Users\Admin\Desktop\sahitya\projects, inifile:

collecting 0 items

collecting 1 item

progs\sample.py ..

========================== 1 passed in 0.11 seconds ===========================

**-v, –verbose**

The -v/--verbose option reports more information than without it. The most obvious difference is that each test gets its own line, and the name of the test and the outcome are spelled out instead of indicated with just a dot.

pytest sample.py -v

rootdir: C:\Users\Admin\Desktop\sahitya\projects, inifile:

collecting 0 items

collecting 1 item

progs/test\_closure.py::sample PASSED

========================== 1 passed in 0.14 seconds ===========================

**-k ,option:**

The -k option lets you use an expression to find what test functions to run. Pretty powerful. It can be used as a shortcut to running an individual test if its name is unique, or running a set of tests that have a common prefix or suffix in their names.

pytest -k "assert"

then only runs programs which has assert in prefix. Remaining all are skipped

**-m MARKEXPR:**

Markers are one of the best ways to mark a subset of your test functions so that they can be run together. As an example, one way to run test\_replace() and test\_member\_access(), even though they are in separate files, is to mark them. You can use any marker name. Let’s say you want to use run\_these\_please. You’d mark a test using the decorator @pytest.mark.run\_these\_please, like so:

import​ pytest ​ ... ​ @pytest.mark.run\_these\_please ​

​def​ test\_member\_access(): ​ ...

pytest -v -m run\_these\_please --- Only file which has that marker will run

**-s and –capture=method :**

The -s flag allows print statements—or really any output that normally would be printed to stdout —to actually be printed to stdout while the tests are running. It is a shortcut for --capture=no. This makes sense once you understand that normally the output is captured on all tests. Failing tests will have the output reported after the test runs on the assumption that the output will help you understand what went wrong. The -s or --capture=no option turns off output capture. When developing tests, I find it useful to add several print() statements so that I can watch the flow of the test.

**Using assert Statements**

When you write test functions, the normal Python assert statement is your primary tool to communicate test failure. The simplicity of this within pytest is brilliant. It’s what drives a lot of developers to use pytest over other frameworks. If you’ve used any other testing framework, you’ve probably seen various assert helper functions. For example, the following is a list of a few of the assert forms and assert helper functions:

pytest unittest

assert something assertTrue(something)

assert a == b assertEqual(a, b)

assert a <= b assertLessEqual(a, b)

… … With pytest, you can use assert with any expression. If the expression would evaluate to False if converted to a bool, the test would fail.

**Overview of unittest:**

The unittest module used to be called PyUnit, due to it’s legacy as a xUnit style framework.  
It works much the same as the other styles of xUnit, and if you’re familiar with unit testing in other languages, this framework (or derived versions), may be the most comfortable for you.

The standard workflow is:  
1. You define your own class derived from unittest.TestCase.  
2. Then you fill it with functions that start with ‘test\_’.  
3. You run the tests by placing unittest.main() in your file, usually at the bottom.

One of the many benifits of unittest, that you’ll use when your tests get bigger than the toy examples I’m showing on this blog, is the use of ‘setUp’ and ‘tearDown’ functions to get your system ready for the tests.

**EX:** test\_sample.py

import unittest

class test\_UT():

def test\_1():...

def test\_2():...

if \_\_name\_\_ == '\_\_main\_\_':

    unittest.main()

**Running unit tests:**

At the bottom of the test file, we have this code:

if \_\_name\_\_ == '\_\_main\_\_':

    unittest.main()

This allows us to run all of the test code just by running the file.  
Running it with no options is the most terse, and running with a ‘-v’ is more verbose, showing which tests ran.

python test\_sample.py

..

----------------------------------------------------------------------

Ran 2 tests in 0.000s

OK

> python test\_um\_unittest.py -v

test\_1 (\_\_main\_\_.test\_UT) ... ok

test\_2 (\_\_main\_\_.test\_UM) ... ok

----------------------------------------------------------------------

Ran 2 tests in 0.000s

OK