There are several testing frameworks available for Python, although only one, unittest comes as part of the typical Python installation. Typical libraries include Unit test, (which is available within the Python distribution by default) and PyTest. In this Topic we will look at PyTest and how it can be used to write unit tests in Python for both functions and classes.

**What Is PyTest?**

PyTest is a testing library for Python; it is currently one of the most popular Python testing libraries (others include unittest and doctest). PyTest can be used for various levels of testing, although its most common application is as a unit testing framework. It is also often used as a testing framework within a TDD based development project. In fact, it is used by Mozilla and Dropbox as their Python testing framework. PyTest offers a large number of features and great flexibility in how tests are written and in how set up behaviour is defined. It automatically finds test based on naming conventions and can be easily integrated into a range of editors and IDEs including PyCharm.

You will probably need to set up PyTest so that you can use it from within your environment. If you are using the PyCharm editor, then you will need to add the PyTest module to the current PyCharm project and tell PyCharm that you want to use PyTest to run all tests for you

PyTest Example

**%%**writefile test\_sample.py

**def** func(x):

**return** x **+** 1

​

**def** func1(x):

**return** x**+**2

Overwriting test\_sample.py

**%%**writefile testing.py

**from** test\_sample **import** func,func1

**import** pytest

​

**def** test\_answer():

**assert** func(3) **==** 4

**def** test\_answer1():

**assert** func1(3) **==** 4

Overwriting testing.py

**!**pytest testing.py

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

platform win32 -- Python 3.7.3, pytest-6.0.2, py-1.9.0, pluggy-0.13.1

rootdir: C:\Users\asha.t\AppData\Local\Programs\Python\Python37-32\Python - Training\15 - Unit testing

collected 2 items

testing.py .F [100%]

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

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ test\_answer1 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

def test\_answer1():

> assert func1(3) == 4

E assert 5 == 4

E + where 5 = func1(3)

testing.py:8: AssertionError

=========================== short test summary info ===========================

FAILED testing.py::test\_answer1 - assert 5 == 4

========================= 1 failed, 1 passed in 0.44s =========================

*#if we supress all the details and quietly execute*

**!**pytest **-**q testing.py

F [100%]

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

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ test\_answer \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

def test\_answer():

> assert func(3) == 5

E assert 4 == 5

E + where 4 = func(3)

testing.py:5: AssertionError

=========================== short test summary info ===========================

FAILED testing.py::test\_answer - assert 4 == 5

1 failed in 0.30s

pytest discovers all tests following its Conventions for Python test discovery, so it finds both test\_ prefixed functions. There is no need to subclass anything, but make sure to prefix your class with Test otherwise the class will be skipped. We can simply run the module by passing its filename

**%%**writefile sample2.py

**class** TestClass:

**def** test\_one(self):

x **=** "this"

**assert** "h" **in** x

​

**def** test\_two(self):

x **=** "hello"

**assert** hasattr(x, "check")

Overwriting sample2.py

**!**pytest sample2.py

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

platform win32 -- Python 3.7.3, pytest-6.0.2, py-1.9.0, pluggy-0.13.1

rootdir: C:\Users\asha.t\AppData\Local\Programs\Python\Python37-32\Python - Training\15 - Unit testing

collected 2 items

sample2.py .F [100%]

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

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ TestClass.test\_two \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

self = <sample2.TestClass object at 0x05C3D170>

def test\_two(self):

x = "hello"

> assert hasattr(x, "check")

E AssertionError: assert False

E + where False = hasattr('hello', 'check')

sample2.py:8: AssertionError

=========================== short test summary info ===========================

FAILED sample2.py::TestClass::test\_two - AssertionError: assert False

========================= 1 failed, 1 passed in 0.39s =========================

To be able to explore PyTest we first need something to test; we will therefore define a simple Calculator class. The calculator keeps a running total of the operations performed; it allows a new value to be set and then this value can be added to, or subtracted from, that accumulated total.

**%%**writefile calculating.py

**class** Calculator:

**def** \_\_init\_\_(self):

self.current **=**0

self.total **=** 0

**def** set(self,value):

self.current **=** value

**def** add(self):

self.total **=** self.total **+** self.current

**def** sub(self):

self.total **=** self.total **-** self.current

**def** total(self):

**return** self.total

​

Overwriting calculating.py

We will now create a very simple PyTest unit test for our Calculator class. This test will be defined in a class called test\_calculator.py. You will need to import the calculator class we wrote above into your test\_calculator.py file We will now define a test, the test should be pre-fixed with test\_ for PyTest to find them. In fact PyTest uses several conventions to find tests, which are: • Search for test\_\*.py or \*\_test.py files. • From those files, collect test items: – test\_prefixed test functions, – test\_prefixed test methods inside Test prefixed test classes (without an\_\_init\_\_method).

**%%**writefile test\_calculator.py

**from** calculating **import** Calculator

**import** pytest

​

**def** test\_add\_one():

calc **=** Calculator() *#create an object so that init will be called and init values will be set*

calc.set(1) *#then set the value to be added to the current attribute*

calc.add()

**assert** calc.total **==** 1 *#No need to learn assertSomething methods unlike other testing frameworks*

Overwriting test\_calculator.py

**!**pytest test\_calculator.py

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

platform win32 -- Python 3.7.3, pytest-6.0.2, py-1.9.0, pluggy-0.13.1

rootdir: C:\Users\asha.t\AppData\Local\Programs\Python\Python37-32\Python - Training\15 - Unit testing

collected 1 item

test\_calculator.py F [100%]

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

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ test\_add\_one \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

def test\_add\_one():

calc = Calculator() #create an object so that init will be called and init values will be set

calc.set(1) #then set the value to be added to the current attribute

calc.add()

> assert calc.total == 2 #No need to learn assertSomething methods unlike other testing frameworks

E assert 1 == 2

E + where 1 = <calculating.Calculator object at 0x06183710>.total

test\_calculator.py:8: AssertionError

=========================== short test summary info ===========================

FAILED test\_calculator.py::test\_add\_one - assert 1 == 2

============================== 1 failed in 0.35s ==============================

The test function creates a new instance of the Calculator class and then calls several methods on it; to set up the value to add, then the call to the add() method itself etc. The final part of the test is the assertion. The assert verifies that the behaviour of the calculator is as expected. The PyTest assert statement works out what is being tested and what it should do with the result—including adding information to be added to a test run report. It avoids the need to have to learn a load of assertSomething type methods (unlike some other testing frameworks).

Note that a test without an assertion is not a test; i.e. it does not test anything. Many IDEs provide direct support for testing frameworks including PyCharm.

We can teststandalone functions **as** well **as** classes using PyTest. For example, given the

function increment below

We can write a PyTest test **for** this **as** follows:

The only real difference **is** that we have **not** had to make an instance of a **class**:

**%%**writefile incre.py

**def** increment(x):

**return** x **+** 1

Writing incre.py

**%%**writefile incremnt\_testing.py

**import** pytest

**from** incre **import** increment

**def** test\_increment\_integer\_3():

**assert** increment(3) **==** 4

Writing incremnt\_testing.py

**!**pytest incremnt\_testing.py

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

platform win32 -- Python 3.7.3, pytest-6.0.2, py-1.9.0, pluggy-0.13.1

rootdir: C:\Users\asha.t\AppData\Local\Programs\Python\Python37-32\Python - Training\15 - Unit testing

collected 1 item

incremnt\_testing.py . [100%]

============================== 1 passed in 0.10s ==============================

**Test Fixtures**

It is not uncommon to need to run some behaviour before or after each test or indeed before or after a group of tests. Such behaviours are defined within what is commonly known as test fixtures. We can add specific code to run:

• at the beginning and end of a test class module of test code (setup\_module/ teardown\_module)

• at the beginning and end of a test class (setup\_class/teardown\_class) or using the alternate style of the class level fixtures (setup/teardown)

• before and after a test function call (setup\_function/teardown\_function)

• before and after a test method call (setup\_method/teardown\_method)

To illustrate why we might use a fixture, let us expand our Calculator test:

**%%**writefile test\_calculator.py

**from** calculating **import** Calculator

**import** pytest

​

**def** test\_initial\_value():

calc **=** Calculator()

**assert** calc.total **==** 0

**def** test\_add\_one():

calc **=** Calculator()

calc.set(1)

calc.add()

**assert** calc.total **==** 1

**def** test\_subtract\_one():

calc **=** Calculator()

calc.set(1)

calc.sub()

**assert** calc.total **==** **-**1

​

**def** test\_add\_one\_and\_one():

calc **=** Calculator()

calc.set(1)

calc.add()

calc.set(1)

calc.add()

**assert** calc.total **==** 2

​

Overwriting test\_calculator.py

**!**pytest test\_calculator.py

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

platform win32 -- Python 3.7.3, pytest-6.0.2, py-1.9.0, pluggy-0.13.1

rootdir: C:\Users\asha.t\AppData\Local\Programs\Python\Python37-32\Python - Training\15 - Unit testing

collected 4 items

test\_calculator.py .... [100%]

============================== 4 passed in 0.20s ==============================

One of the issues with this set of tests is that we have repeated the creation of the Calculator object at the start of each test. While this is not a problem in itself it does result in duplicated code and the possibility of future issues in terms of maintenance if we want to change the way a calculator is created. It may also not be as efficient as reusing the Calculator object for each test. We can however, define a fixture that can be run before each individual test function is executed. To do this we will write a new function and use the pytest.fixture decorator on that function. This marks the function as being special and that it can be used as a fixture on an individual function. Functions that require the fixture should accept a reference to the fixture as an argument to the individual test function. For example, for a test to accept a fixture called calculator; it should have an argument with the fixture name, i.e. calculator. This name can then be used to access the object returned.

**%%**writefile test\_calculator.py

**from** calculating **import** Calculator

**import** pytest

​

@pytest.fixture

**def** calculator():

**return** Calculator()

​

**def** test\_initial\_value(calculator):

**assert** calculator.total **==** 0

**def** test\_add\_one(calculator):

calculator.set(1)

calculator.add()

**assert** calculator.total **==** 1

**def** test\_subtract\_one(calculator):

calculator.set(1)

calculator.sub()

**assert** calculator.total **==** **-**1

**def** test\_add\_one\_and\_one(calculator):

calculator.set(1)

calculator.add()

calculator.set(1)

calculator.add()

**assert** calculator.total **==** 2

Overwriting test\_calculator.py

**!**pytest test\_calculator.py

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

platform win32 -- Python 3.7.3, pytest-6.0.2, py-1.9.0, pluggy-0.13.1

rootdir: C:\Users\asha.t\AppData\Local\Programs\Python\Python37-32\Python - Training\15 - Unit testing

collected 4 items

test\_calculator.py .... [100%]

============================== 4 passed in 0.12s ==============================

​

In the above code, each of the test functions accepts the calculator fixture that is used to instantiate the Calculator object. We have therefore de-duplicated our code; there is now only one piece of code that defines how a calculator object should be created for our tests. Note each test is supplied with a completely new instance of the Calculator object; there is therefore no chance of one test impacting on another test.

The PyTest fixtures can be applied to functions (as above), classes, modules, packages or sessions. The scope of a fixture can be indicated via the (optional) scope parameter to the fixture decorator. The default is “function” which is why we did not need to specify anything above. The scope determines at what point a fixture should be run. For example, a fixture with ‘session’ scope will be run once for the test session, a fixture with module scope will be run once for the module (that is the fixture and anything it generates will be shared across all tests in the current module), a fixture with class scope indicates a fixture that is run for each new instance of a test class created etc.

Another parameter to the fixture decorator is autouse which if set to True will activatethe fixture for all tests that can see it. If it is settoFalse(which is the default) then an explicit reference in a test function (or method etc.) is required to activate the fixture.

**%%**writefile test\_calculator.py

**from** calculating **import** Calculator

**import** pytest

​

@pytest.fixture(scope**=**'session', autouse**=True**)

**def** session\_scope\_fixture():

print('session\_scope\_fixture')

@pytest.fixture(scope**=**'module', autouse**=True**)

**def** module\_scope\_fixture():

print('module\_scope\_fixture')

​

@pytest.fixture(scope**=**'class', autouse**=True**)

**def** class\_scope\_fixture():

print('class\_scope\_fixture')

@pytest.fixture

**def** calculator():

**return** Calculator()

​

​

**def** test\_initial\_value(calculator):

**assert** calculator.total **==** 0

**def** test\_add\_one(calculator):

calculator.set(1)

calculator.add()

**assert** calculator.total **==** 2

**def** test\_subtract\_one(calculator):

calculator.set(1)

calculator.sub()

**assert** calculator.total **==** 1

​

**def** test\_add\_one\_and\_one(calculator):

calculator.set(1)

calculator.add()

calculator.set(1)

calculator.add()

**assert** calculator.total **==** 3

Overwriting test\_calculator.py

**!**pytest test\_calculator.py

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

platform win32 -- Python 3.7.3, pytest-6.0.2, py-1.9.0, pluggy-0.13.1

rootdir: C:\Users\asha.t\AppData\Local\Programs\Python\Python37-32\Python - Training\15 - Unit testing

collected 4 items

test\_calculator.py .FFF [100%]

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

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ test\_add\_one \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

calculator = <calculating.Calculator object at 0x057B7610>

def test\_add\_one(calculator):

calculator.set(1)

calculator.add()

> assert calculator.total == 2

E assert 1 == 2

E + where 1 = <calculating.Calculator object at 0x057B7610>.total

test\_calculator.py:27: AssertionError

---------------------------- Captured stdout setup ----------------------------

class\_scope\_fixture

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ test\_subtract\_one \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

calculator = <calculating.Calculator object at 0x057939F0>

def test\_subtract\_one(calculator):

calculator.set(1)

calculator.sub()

> assert calculator.total == 1

E assert -1 == 1

E + where -1 = <calculating.Calculator object at 0x057939F0>.total

test\_calculator.py:31: AssertionError

---------------------------- Captured stdout setup ----------------------------

class\_scope\_fixture

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ test\_add\_one\_and\_one \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

calculator = <calculating.Calculator object at 0x057935F0>

def test\_add\_one\_and\_one(calculator):

calculator.set(1)

calculator.add()

calculator.set(1)

calculator.add()

> assert calculator.total == 3

E assert 2 == 3

E + where 2 = <calculating.Calculator object at 0x057935F0>.total

test\_calculator.py:38: AssertionError

---------------------------- Captured stdout setup ----------------------------

class\_scope\_fixture

=========================== short test summary info ===========================

FAILED test\_calculator.py::test\_add\_one - assert 1 == 2

FAILED test\_calculator.py::test\_subtract\_one - assert -1 == 1

FAILED test\_calculator.py::test\_add\_one\_and\_one - assert 2 == 3

========================= 3 failed, 1 passed in 0.42s =========================

**Parameterised Tests**

One common requirement of a test to run the same tests multiple times with several different input values. This can greatly reduce the number of tests that must be defined. Such tests are referred to as parametrised tests; with the parameter values for the test specified using the @pytest.mark.parametrize decorator.

**%%**writefile test\_calculator.py

**from** calculating **import** Calculator

**import** pytest

​

@pytest.fixture

**def** calculator():

**return** Calculator()

​

​

@pytest.mark.parametrize('input1,input2,expected', [(3, 1, 4), (3, 2, 5),])

**def** test\_calculator\_add\_operation(calculator, input1,input2,expected):

calculator.set(input1)

calculator.add()

calculator.set(input2)

calculator.add()

**assert** calculator.total **==** expected

Overwriting test\_calculator.py

**!**pytest test\_calculator.py

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

platform win32 -- Python 3.7.3, pytest-6.0.2, py-1.9.0, pluggy-0.13.1

rootdir: C:\Users\asha.t\AppData\Local\Programs\Python\Python37-32\Python - Training\15 - Unit testing

collected 2 items

test\_calculator.py .. [100%]

============================== 2 passed in 0.12s ==============================

This illustrates setting up a parametrised test for the Calculator in which two input values are added together and compared with the expected result. Note that the parameters are named in the decorator and then a list of tuples is used to define the values to be used for the parameters. In this case the test\_ calculator\_add\_operation will be run two passing in 3, 1 and 4 and then passing in 3, 2 and 5 for the parameters input1, input2 and expected respectively.

**Testing for Exceptions**

To verify the presence of an exception in PyTest use the with statement and pytest.raises. This is a context manager that will verify on exit that the specified exception was raised. It is used as follows:

with pytest.raises(accounts.BalanceError):

current\_account.withdraw(200.0)

**Ignoring Tests**

In some cases it is useful to write a test for functionality that has not yet been implemented; this may be to ensure that the test is not forgotten or because it helps to document what the item under test should do. However, if the test is run then the test suite as a whole will fail because the test is running against behaviour that has yet to be written One way to address this problem is to decorate a test with the @pytest.- mark.skip decorator:

**%%**writefile test\_calculator.py

**from** calculating **import** Calculator

**import** pytest

​

@pytest.fixture

**def** calculator():

**return** Calculator()

​

​

@pytest.mark.parametrize('input1,input2,expected', [(3, 1, 4), (3, 2, 5),])

**def** test\_calculator\_add\_operation(calculator, input1,input2,expected):

calculator.set(input1)

calculator.add()

calculator.set(input2)

calculator.add()

**assert** calculator.total **==** expected

​

@pytest.mark.skip(reason**=**'not implemented yet')

**def** test\_calculator\_multiply(calculator):

calculator.multiply(2, 3)

**assert** calculator.total **==** 6

Overwriting test\_calculator.py

**!**pytest test\_calculator.py

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

platform win32 -- Python 3.7.3, pytest-6.0.2, py-1.9.0, pluggy-0.13.1

rootdir: C:\Users\asha.t\AppData\Local\Programs\Python\Python37-32\Python - Training\15 - Unit testing

collected 3 items

test\_calculator.py ..s [100%]

======================== 2 passed, 1 skipped in 0.12s =========================