# Chapter 15
# PyTest Testing Framework


## 15.1 Introduction

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 chapter we will look at PyTest and how it can be used to write unit tests in Python for both functions and classes.

## 15.2 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.



J. Hunt, *Advanced Guide to Python 3 Programming*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-030-25943-3_15

## 15.3 Setting Up PyTest

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.

## 15.4 A Simple PyTest Example

### Something to Test

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.

```
class Calculator:
    def __init__(self):
        self.current = 0
        self.total = 0

    def set(self, value):
        self.current = value

    def add(self):
        self.total += self.current

    def sub(self):
        self.total -= self.current

    def total(self):
        return self.total
```

Save this class into a file called `calculator.py`.

### Writing a Test

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 (remember each file is a module in Python).

The exact `import` statement will depend on where you placed the calculator file relative to the test class. In this case the two files are both in the same directory and so we can write:

```
from calculator import Calculator
```

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).

Note that we keep test files and the files containing the code to be tested separate; indeed in many cases they are kept in different directory structures. This means that there is not chance of developers accidentally using tests in production code etc.

Now we will add to the file a function that defines a test. We will call the function `test_add_one`; it needs to start with `test_` due to the above convention. However, we have tried to make the rest of the function name descriptive, so that its clear what it is testing. The function definition is given below:

```
from calculator import Calculator

def test_add_one():
    calc = Calculator()
    calc.set(1)
    calc.add()
    assert calc.total == 1
```

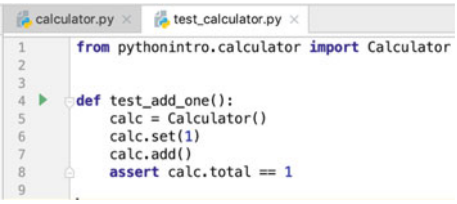
The test function creates a new instance of the `Calculator` class and then calls several methods on it; to set up the value to add, then the call to the `add()` method itself etc.

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).

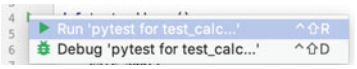
Note that a test without an assertion is *not* a test; i.e. it does not test anything.

Many IDEs provide direct support for testing frameworks including PyCharm. For example, PyCharm will now detect that you have written a function with an `assert` statement in it and add a *Run Test* icon to the grey area to the left of the

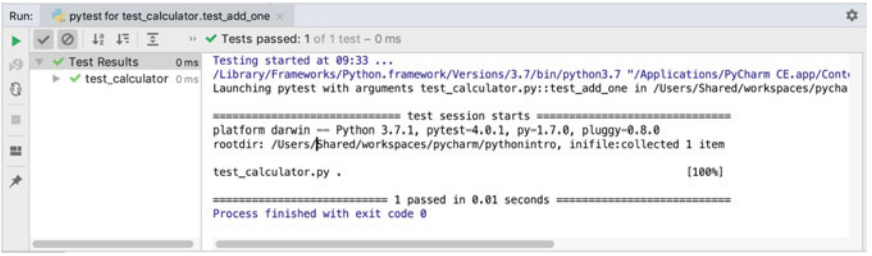
editor. This can be seen in the following picture where a green arrow has been added at line 4; this is the 'Run Test' button:




The developer can click on the green arrow to run the test. They will then be presented with the *Run* menu that is preconfigured to use PyTest for you:




If the developer now selects the Run option; this will use the PyTest runner to execute the test and collect information about what happened and present it in a PyTest output view at the bottom of the IDE:

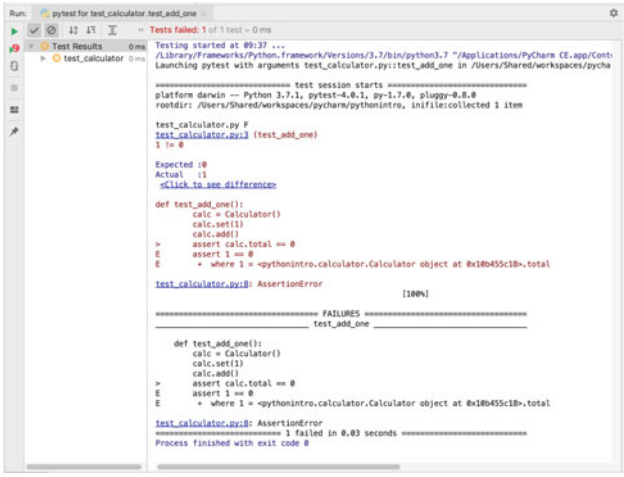



Here you can see a tree in the left-hand panel that currently holds the one test defined in the `test_calculator.py` file. This tree shows whether tests have passed or failed. In this case we have a green tick showing that the test passed.

To the right of this tree is the main output panel which shows the results of running the tests. In this case it shows that PyTest ran only one test and that this was the `test_add_one` test which was defined in `test_calculator.py` and that 1 test passed.

If you now change the assertion in the test to check to see that the result is 0 the test will fail. When run, the IDE display will update accordingly.

The tree in the left-hand pane now shows the test as failed while the right-hand pane provides detailed information about the test that failed including where in the test the failed assertion was defined. This is very helpful when trying to debug test failures.




## 15.5 Working with PyTest

### Testing Functions

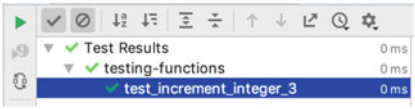
We can test *standalone* functions as well as classes using PyTest. For example, given the function `increment` below (which merely adds one to any number passed into it):

```
def increment(x):
    return x + 1
```

We can write a PyTest test for this as follows:

```
def test_increment_integer_3():
    assert increment(3) == 4
```

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




### Organising Tests

Tests can be grouped together into one or more files; PyTest will search for all files following the naming convention (file names that either start or end with '`test`') in specified locations:

- If no arguments are specified when PyTest is run then the search for suitably named test files starts from the `testpaths` environment variable (if configured) or the current directory. Alternatively, command line arguments can be used in any combination of directories or filenames etc.

- PyTest will recursively search down into sub directories, unless they match `norecursedirs` environment variable.
- In those directories, it will search for files that match the naming conventions `test_*.py` or `*_test.py` files.

Tests can also be arranged within test files into Test classes. Using test classes can be helpful in grouping tests together and managing the setup and tear down behaviours of separate groups of tests. However, the same effect can be achieved by separating the tests relating to different functions or classes into different files.

**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:

```
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
```

We now have four tests to run (we could go further but this is enough for now).

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. This is illustrated below:

```
import pytest
from calculator import Calculator

@pytest.fixture
def calculator():
    """Returns a Calculator instance"""
    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
```

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.

It is also considered good practice to add a *docstring* to your fixtures as we have done above. This is because PyTest can produce a list of all fixtures available along with their docstrings. From the command line this is done using:

```
> pytest fixtures
```

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 activate the fixture for all tests that can see it. If it is set to `False` (which is the default) then an explicit reference in a test function (or method etc.) is required to activate the fixture.

If we add some additional fixtures to our tests we can see when they are run:

```
import pytest
from calculator import Calculator

  @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():
     """Returns a Calculator instance"""
     print('calculator fixture')
     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
```

If we run this version of the tests, then the output shows when the various fixtures are run:

```
session_scope_fixture
module_scope_fixture
class_scope_fixture
calculator fixture
.class_scope_fixture
calculator fixture
.class_scope_fixture
calculator fixture
.class_scope_fixture
calculator fixture
```

Note that higher scoped fixtures are instantiated first.

## 15.6 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.

**@pytest.mark.parametrize** decorator.

```
@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
```

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

You can write tests that verify that an exception was raised. This is useful as testing negative behaviour is as important as testing positive behaviour. For example, we might want to verify that a particular exception is raised when we attempt to withdraw money from a bank account which will take us over our overdraft limit.

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)
```

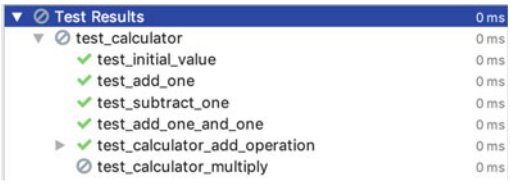
### Ignoring Tests

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.

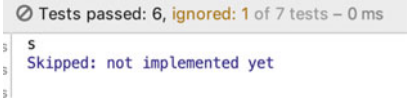
One way to address this problem is to decorate a test with the `@pytest.mark.skip` decorator:

```
@pytest.mark.skip(reason='not implemented yet')
def test_calculator_multiply(calculator):
    calculator.multiply(2, 3)
    assert calculator.total == 6
```

This indicates that PyTest should record the presence of the test but should not try to execute it. PyTest will then note that the test was skipped, for example in PyCharm this is shown using a circle with a line through it.




It is generally considered best practice to provide a reason why the test has been skipped so that it is easier to track. This information is also available when PyTest skips the test:




## 15.7 Online Resources

See the following online resources for information on PyTest:

- http://pythontesting.net/framework/PyTest/PyTest-introduction/PyTest introduction.
- https://github.com/pluralsight/intro-to-PyTest An example based introduction to PyTest.
- https://docs.pytest.org/en/latest/PyTest home page.
- https://docs.pytest.org/en/latest/#documentation PyTest documentation.

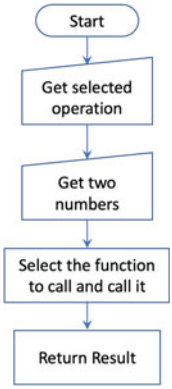
## 15.8 Exercises

Create a simple `Calculator` class that can be used for testing purposes. This simple calculator can be used to add, subtract, multiple and divide numbers.

This will be a purely command driven application that will allow the user to specify

- the operation to perform and
- the two numbers to use with that operation.

The `Calculator` object will then return a result. The same object can be used to repeat this sequence of steps. This general behaviour of the `Calculator` is illustrated below in flow chart form:




You should also provide a memory function that allows the current result to be added to or subtracted from the current memory total. It should also be possible to retrieve the value in memory and clear the memory.

Next write a PyTest set of tests for the `Calculator` class.

Think about what tests you need to write; remember you can't write tests for every value that might be used for an operation; but consider the boundaries, 0, −1, 1, −10, +10 etc.

Of course you also need to consider the cumulative effect of the behaviour of the memory feature of the calculator; that is multiple memory adds or memory subtractions and combinations of these.

As you identify tests you may find that you have to update your implementation of the `Calculator` class. Have you taken into account all input options, for example dividing by zero—what should happen in these situations.