Week 7
Unit Testing

In this week's note, we cover following topics.

- Automated Unit testing.
- PyUnit Test Framework.
- How to install PyUnit.
- What are the basics of unit testing using PyUnit and how to create unit tests?
- How to use Command Line and GUI (through IDE) to run and execute test cases.
- What methods are used or useful while writing test cases and some rules.
- How to calculate and design coverages using Unit Tests.

Introduction

A well-designed large software system will be composed of many units - small, relatively self-contained chunks of related functionality, available through a well-designed interface which controls the ways that the unit interacts with the rest of the system. In an object-oriented design, units are usually classes.

Because they are small and relatively self-contained, debugging them in isolation is much easier and quicker than trying to debug in the context of many interacting units. So, it is best to find as many problems in isolation as possible. To do that, each unit should have unit tests to exercise the functionality of that unit.

Unit tests are typically written by the developers of that unit and are almost always automated.

While unit testing is incredibly useful, we also need to test the integration of that unit with the larger system. This process is called, unsurprisingly, integration testing. The trick in integration testing is what bits of the larger system you integrate with. In the last couple of decades or so, there has been a revolution in how you do this, and continuous integration is now the standard way you integrate units into the broader system.

- First make sure that the individual units are working.
- Test that combine these units (integration testing)

Automated Unite Testing

Unit testing has been carried out, and automated, for as long as the concept of modularization in software has existed. However, the practice of unit testing was given a major kick along by Kent Beck, who developed a unit testing framework for the Smalltalk programming language called sUnit[1] in the 1980s. This was adapted by Ehrich Gamma (one of the authors of the highly influential software design book Design Patterns) for jUnit[2], a Java test framework with the same basic design. Implementations

[1] https://sunit.sourceforge.net/
[2] https://junit.org/junit5/

in many other languages have now been created and generally follow the "XUnit" naming convention; Wikipedia has a nice list of unit testing frameworks[3], both XUnit and otherwise.

PyUnit Test Framework

Testing frameworks enables us to write our test cases in a way that they can be easily executed by the machine. The Python unit testing framework, sometimes referred to as “PyUnit,” is a Python language version of JUnit developed by Kent Beck and Erich Gamma. PyUnit forms part of the Python Standard Library as of Python version 3.3. It is not the only unit test tool for Python - a module called nose has some nice features. Additionally, PyTest is another useful Unit Testing module. PyUnit because of its commonality with other testing systems we will demonstrate it here.

The Python unit testing framework supports test automation, sharing of setup and shutdown code for tests, aggregation of tests into collections, and independence of the tests from the reporting framework. The unittest module provides classes that make it easy to support these qualities for a set of tests.

- Write test code in a separate file than the production code.
- The production code will be shipped without the test code.
- Do not send test code to the client unless it is a part of the contract.

Why use PyUnit?

Here is the question why we should use PyUnit and not make our own framework. First and foremost, why reinvent a wheel when someone has done everything for us. These frameworks are not hard to use and gradually you will get used to it. It will be time consuming to write and set up our own framework for testing. In an industrial environment this will be time consuming and tedious as we are short of time and money. Most importantly, we have to test harness that loads/runs tests, compares the expected output with actual output, shows messages for any failing tests and summarises the results.

Installing PyUnit

The classes needed to write tests are to be found in the `'unittest'` module. If you are using older versions of Python (prior to Python 2.1), the module can be downloaded from http://pyunit.sourceforge.net/. However, the unittest module is now a part of the standard Python distribution; hence it requires no separate installation. We expect you all to use Python 3.6+ version in this unit.

The Basics

'`unittest`' supports test automation, sharing of setup and shutdown code for tests, aggregation of tests into collections, and independence of the tests from the reporting framework. The unittest module provides classes that make it easy to support these qualities for a set of tests.

To achieve this, unittest supports the following important concepts:

[3] https://en.wikipedia.org/wiki/List_of_unit_testing_frameworks

test fixture - This represents the preparation needed to perform one or more tests, and any associate clean up actions. This may involve, for example, creating temporary or proxy databases, directories, or starting a server process.

test case - This is the smallest unit of testing. This checks for a specific response to a particular set of inputs. `unittest` provides a base class, TestCase, which may be used to create new test cases.

test suite - This is a collection of test cases, test suites, or both. This is used to aggregate tests that should be executed together. Test suites are implemented by the TestSuite class.

test runner - This is a component which orchestrates the execution of tests and provides the outcome to the user. The runner may use a graphical interface, a textual interface, or return a special value to indicate the results of executing the tests.

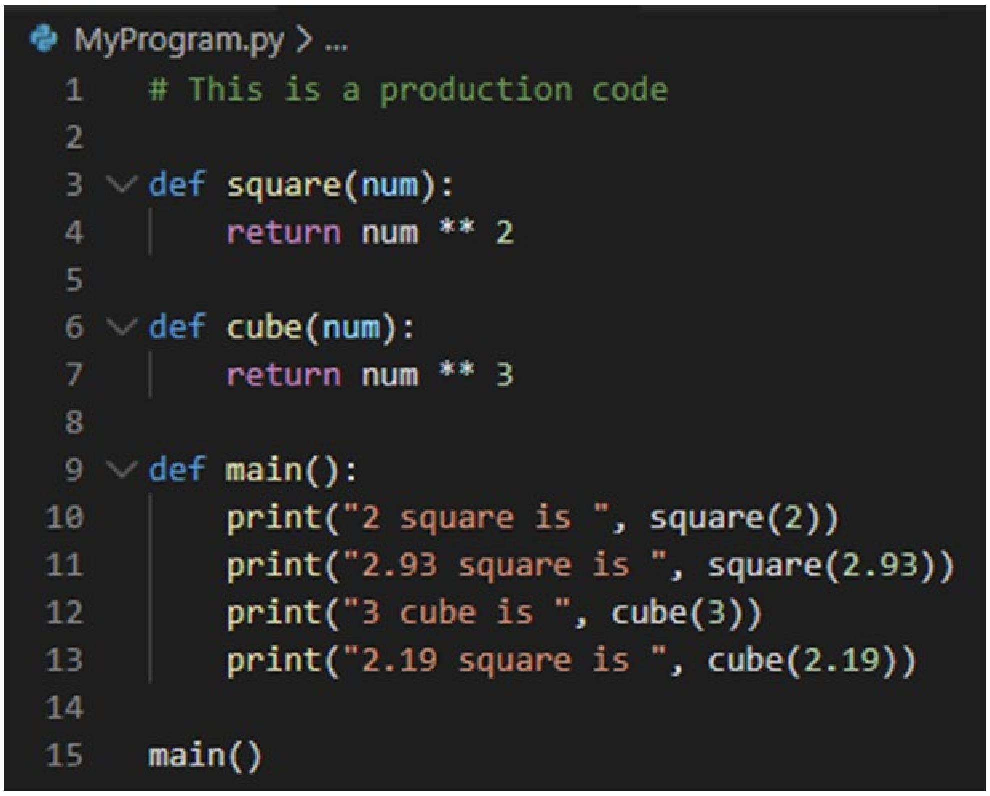
Creating a Unit Test in PyUnit

Testcases and Testsuite are written using unittest module follows a common format

- Define a class derived from `unittest.TestCase()`. We discuss TestCase() in more detail in subsequent sections.
- Define testcases having the nomenclature test_name
- Execute the testcases/testsuites by having `main()`- sometimes referred as `unittest.main()`, which is normally placed at the bottom of the file i.e. after the necessary classes for testcase execution have been implemented.

Example

A simple class calculating squares and cubes of numbers.



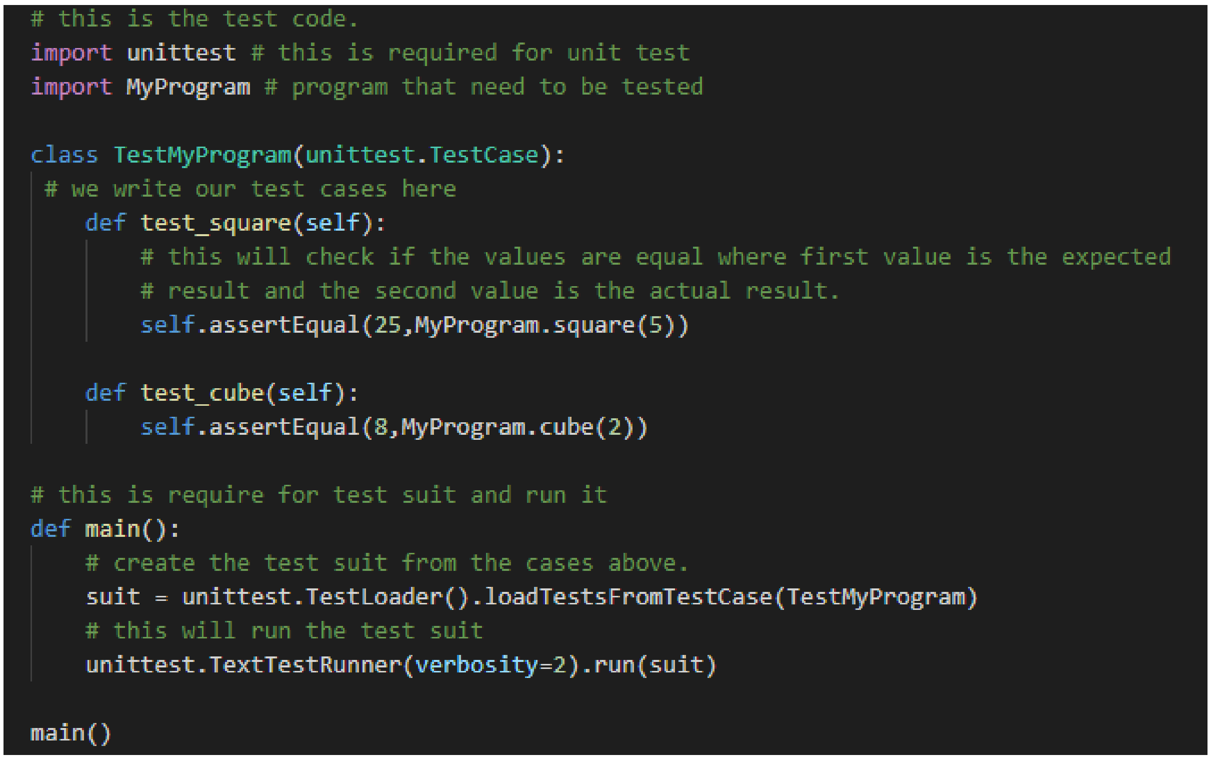
```python
# This is a production code

def square(num):
    return num ** 2

def cube(num):
    return num ** 3

def main():
    print("2 square is ", square(2))
    print("2.93 square is ", square(2.93))
    print("3 cube is ", cube(3))
    print("2.19 square is ", cube(2.19))

main()
```

Figure 1: A simple Python programme calculating squares and cubes of numbers.

We write test cases to verify if the functionality conforms the requirements i.e. the square calculates the square while the cube measures the cube of numbers. Here are some rules to follow while writing unit tests.

- Define the class with the test in it, say TestMyProgram. Remember to make it a separate class and do not write test cases in the production code.
- Import a unittest package (must be the first statement)
- Import the name of the class that needs to be tested. It is MyProgram as in the example above.
- We use the object-oriented way of writing the code so we must declare a class.
- The main method should have test loader that creates the test suit from the test written earlier in the class part
- Next line in the main method should be the run method that runs the suit
- In the class section, we write the test case.

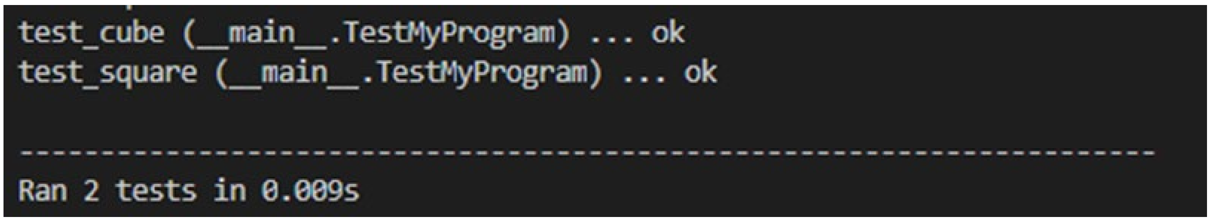
The code looks like this



```python
# this is the test code.
import unittest # this is required for unit test
import MyProgram # program that need to be tested

class TestMyProgram(unittest.TestCase):
 # we write our test cases here
    def test_square(self):
        # this will check if the values are equal where first value is the expected
        # result and the second value is the actual result.
        self.assertEqual(25,MyProgram.square(5))

    def test_cube(self):
        self.assertEqual(8,MyProgram.cube(2))

# this is require for test suit and run it
def main():
    # create the test suit from the cases above.
    suit = unittest.TestLoader().loadTestsFromTestCase(TestMyProgram)
    # this will run the test suit
    unittest.TextTestRunner(verbosity=2).run(suit)

main()
```

Figure 2: Unit Test cases for square and cube methods.

On executing the testing cases, the output looks like this.



```
test_cube (__main__.TestMyProgram) ... ok
test_square (__main__.TestMyProgram) ... ok

----------------------------------------------------------------------
Ran 2 tests in 0.009s
```

Figure 3: the output of test cases.

Running tests using Command Line/Terminal

Normally, I use a command line interface such as Terminal on Linux machines or Command Prompt on Windows to run the Python code and Test cases. Python should be installed on your machine and Python path must be set in the environment variables for the Windows Operating System. For Linux and MacOS python is installed by default.

Following commands are used to run test cases:

```
Python MyProgramTests.py
```

If you have multiple versions of Python installed on your machine, you may need to set up how the Python command works. For Python 2.7, as mentioned earlier, the unit test module is installed separately, and based on your python configuration. As I have installed both Python 2.7 and 3.8 on my Linux machine, so I tested the code by writing the following command.

```
Python3 MyProgramTests.py
```

In case, if the main method is not written in the test code, we can use a unittest hook to run and test the code.

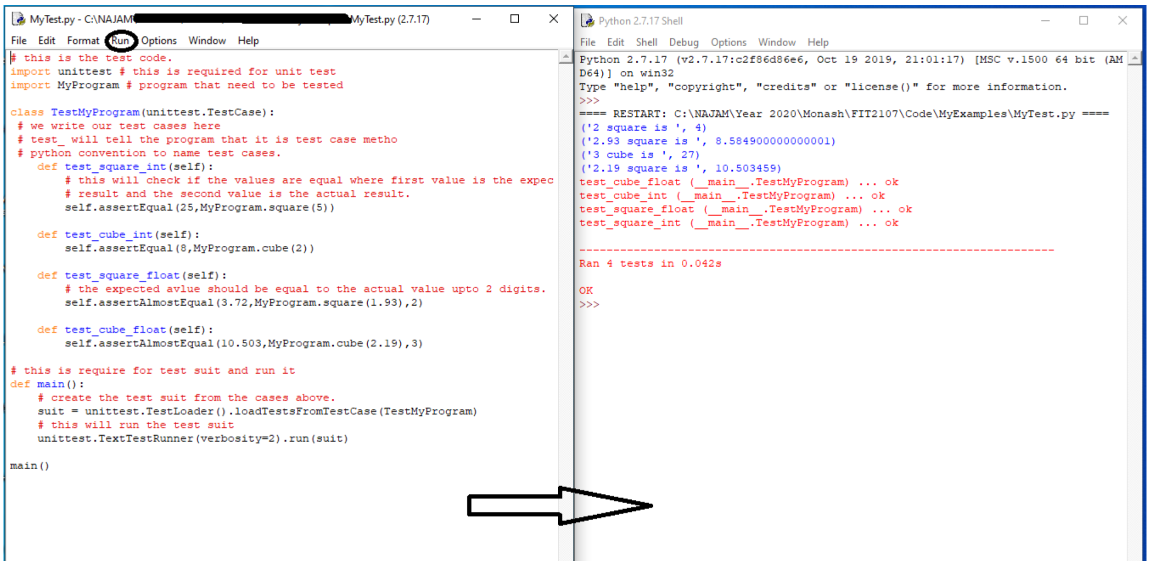
```
python3 -m unittest MyProgramTests.py
```

Running Tests using IDEs

Mainly, I use VisualStudioCode for all types of coding exercises. However, PyCharm is the most commonly used IDE for writing and testing python code. However, here I will show first how can you run a Python Programme as well as the Test suite using Python GUI Shell, which is installed by default on Windows Machine.

Python IDLE

Python has its own GUI shell which is installed by default on Windows OS when you install Python. Seeing the picture below, I have opened the same file in Python 2.7.X distribution installed on my windows machine. I have separately installed unittest framework in Python 2.7. Pressing the run button as highlighted will run your code as well as the test cases. The shortcut key for run is F5.


MyTest.py - C:\NAJAM[redacted]MyTest.py (2.7.17)
File Edit Format Run Options Window Help
Python 2.7.17 Shell
File Edit Shell Debug Options Window Help

```
# this is the test code.
import unittest # this is required for unit test
import MyProgram # program that need to be tested

class TestMyProgram(unittest.TestCase):
 # we write our test cases here
 # test_ will tell the program that it is test case metho
 # python convention to name test cases.
    def test_square_int(self):
        # this will check if the values are equal where first value is the expec
        # result and the second value is the actual result.
        self.assertEqual(25,MyProgram.square(5))

    def test_cube_int(self):
        self.assertEqual(8,MyProgram.cube(2))

    def test_square_float(self):
        # the expected avlue should be equal to the actual value upto 2 digits.
        self.assertAlmostEqual(3.72,MyProgram.square(1.93),2)

    def test_cube_float(self):
        self.assertAlmostEqual(10.503,MyProgram.cube(2.19),3)

# this is require for test suit and run it
def main():
    # create the test suit from the cases above.
    suit = unittest.TestLoader().loadTestsFromTestCase(TestMyProgram)
    # this will run the test suit
    unittest.TextTestRunner(verbosity=2).run(suit)

main()
```

```
Python 2.7.17 (v2.7.17:c2f86d86e6, Oct 19 2019, 21:01:17) [MSC v.1500 64 bit (AM
D64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
==== RESTART: C:\NAJAM\Year 2020\Monash\FIT2107\Code\MyExamples\MyTest.py ====
('2 square is ', 4)
('2.93 square is ', 8.584900000000001)
('3 cube is ', 27)
('2.19 square is ', 10.503459)
test_cube_float (__main__.TestMyProgram) ... ok
test_cube_int (__main__.TestMyProgram) ... ok
test_square_float (__main__.TestMyProgram) ... ok
test_square_int (__main__.TestMyProgram) ... ok

----------------------------------------------------------------------
Ran 4 tests in 0.042s

OK
>>>
```

Figure 4: Python GUI IDLE showing test cases and their output.

Visual Studio Code

In VSCode, you can run the python program and the test cases in two ways.

- This is the same as the command line approach discussed above. VS Code, has a built in terminal (same as command prompt on Windows machines or terminal on Linux). You can navigate through the desired folder and run the command line arguments to run the code. I assume you all are familiar with commands how to navigate through folders on DOS or Linux Terminal so I am skipping it.
- Press the run button as shown in the figure. Run -> Run without Debugging (Ctrl + F5) as shown below

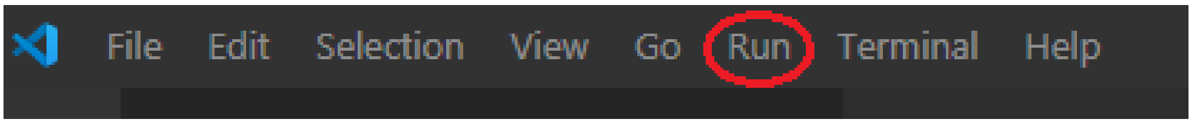
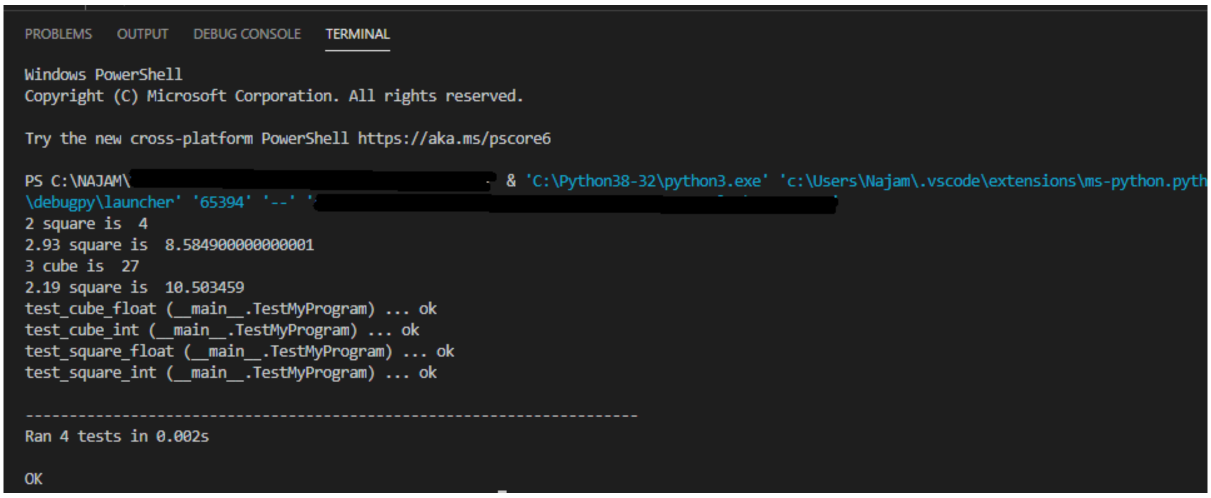


Figure 5: The run button to execute the code in VSCode

If the terminal is not automatically opened when you load the folder or file in VSCode, the run command will open it for us as shown below.



```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\NAJAM\[redacted] & 'C:\Python38-32\python3.exe' 'c:\Users\Najam\.vscode\extensions\ms-python.pyth
\debugpy\launcher' '65394' '--' '[redacted]
2 square is  4
2.93 square is  8.584900000000001
3 cube is  27
2.19 square is  10.503459
test_cube_float (__main__.TestMyProgram) ... ok
test_cube_int (__main__.TestMyProgram) ... ok
test_square_float (__main__.TestMyProgram) ... ok
test_square_int (__main__.TestMyProgram) ... ok

----------------------------------------------------------------------
Ran 4 tests in 0.002s

OK
```

Figure 6: The output of test case in a VSCode terminal

Online IDE

If you want to use an online IDE https://repl.it/ is a good choice.

Outcome of Test Cases

There could be three possible outcomes of the test case

- OK/PASSED: this means your test is passed.
- FAIL: The test does not pass and raises an exception which is AssertionError in most cases.
- ERROR: The test raises an exception other than AssertionError.

Methods used in UnitTest Cases

`Unittest` suit itself contain many methods and going through the official documentation is the best way to explore them. The official documentation is accessible at the following url1https://docs.python.org/3/library/unittest.html. However, I mention a few commonly used methods here.

assertEqual(a,b): This statement is used to check if the result obtained is equal to the expected result.

assertNotEqual(a,b): Ensures that two expressions a and b do not yield the same result.

assertAlmostEqual(a,b,digit): Ensures that two expressions a and b yield the same result to at least this number of digits. It is useful when used with floating points numbers.self.assertAlmostEqual(3.72, MyProgram.square(1.93),2).

assertTrue(x): ensures that the expression yields the results True. It is same as assertEqual(True,a)

assertFalse(x): ensures that the statement yields the result False. It is same as assertEqual(False,a)

assertRaises(): This statement is used to raise a specific exception.

Things to remember while writing test cases

Here are some points that you all keep in mind or follow while writing test cases. Most of these points are based on my experience writing test cases – there may be some variations as it depends on the developers own style as well.

- A testing unit should focus on one tiny bit of functionality (i.e. one unit) and prove it correct. In other words, there must be a test case for each functionality of the code regardless of how many test cases you write for a single method.
- Each test unit must be fully independent. Each test must be able to run alone, and also within the test suite, regardless of the order that they are called. The implication of this rule is that each test must be loaded with a fresh dataset and may have to do some clean-up afterwards. This is usually handled by `setUp()` and `tearDown()` methods.
- Try hard to make tests that run fast. If one single test needs more than a few milliseconds to run, development will be slowed down, or the tests will not be run as often as is desirable. In some cases, tests cannot be fast because they need a complex data structure to work on, and this data structure must be loaded every time the test runs. Keep these heavier tests in a separate test suite that is run by some scheduled task, and run all other tests as often as needed.
- Learn your tools and learn how to run a single test or a test case. Then, when developing a function inside a module, run this function's tests frequently, ideally automatically when you save the code.
- Always run the full test suite before a coding session and run it again after. This will give you more confidence that you did not break anything in the rest of the code.
- It is a good idea to implement a hook that runs all tests before pushing code to a shared repository. We shall discuss how continuous integration works later in week 9.
- Use long and descriptive names for testing functions. The style guide here is slightly different than that of running code, where short names are often preferred. The reason is testing functions are never called explicitly. `square()` or even `sqr()` is ok in running code, but in testing code you would have names such as `test_square_of_number_2()`,

`test_square_negative_number()`. These function names are displayed when a test fails and should be as descriptive as possible.

- Always write meaningful comments for the functionality you perform in a code as well as in a test case. That will help us as well as your colleagues in understanding your functionality.
- When something goes wrong or has to be changed, and if your code has a good set of tests, you or other maintainers will rely largely on the testing suite to fix the problem or modify a given behaviour. Therefore, the testing code will be read as much as or even more than the running code. A unit test whose purpose is unclear is not very helpful in this case.
- Another use of the testing code is as an introduction to new developers. When someone will have to work on the code base, running and reading the related testing code is often the best thing that they can do to start. They will or should discover the hot spots, where most difficulties arise, and the corner cases. If they have to add some functionality, the first step should be to add a test to ensure that the new functionality is not already a working path that has not been plugged into the interface.

Code Coverage in Python

Coverage measurement is typically used to gauge the effectiveness of tests. It can show which parts of your code are being exercised by tests, and which are not. We have talked about coverage in White box testing techniques. - here we learn how we can get code coverage using Python. Python provides a third-party tool called coverage.py for this purpose. Let’s explore it.

Coverage.py

Coverage.py is a third-party tool for measuring code coverages in Python programmes. It provides very nice command line and HTML output along with advanced features such as branch coverage.

Installing coverage.py

Use `pip install coverage` (or pip3) to install coverage for your environment. The best practice is to create a virtual environment first and then run the `pip install coverage` command. However, it will work without creating a virtual environment as well. We normally create a virtual environment to separate testing from real deployments. I will talk about the virtual environments in Week 9.

Commands

As the coverage is installed by now, following command is used to test the coverage through command line.

```
coverage run [Python File Name].py
```

To get the coverage report use.

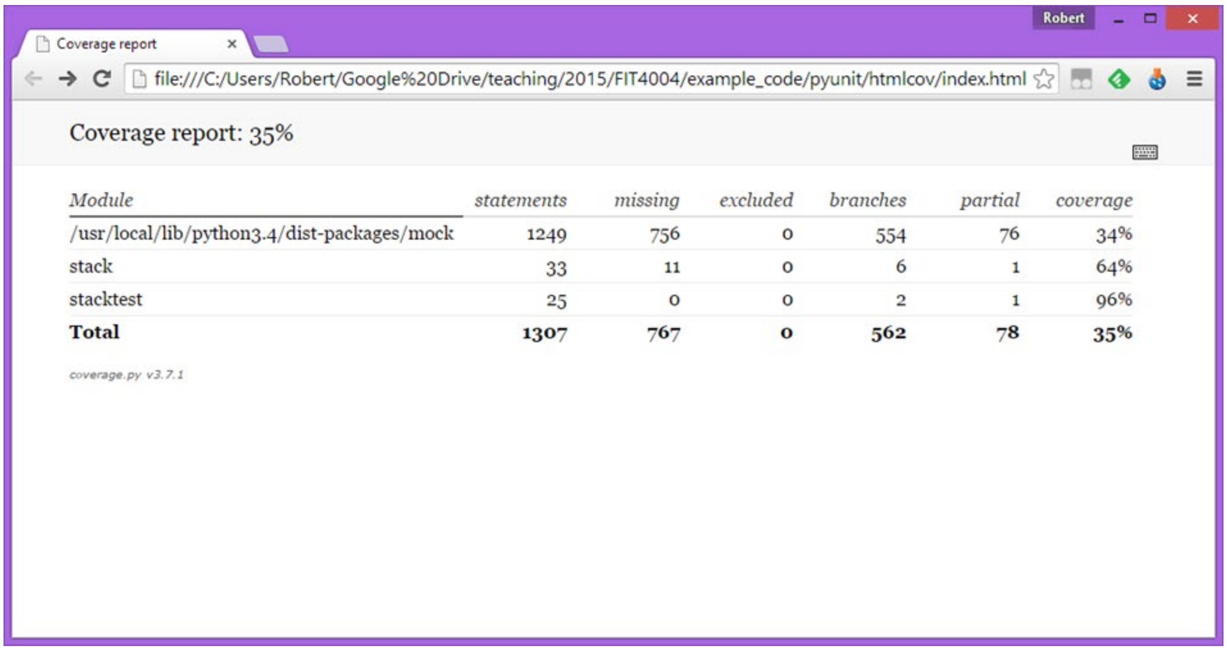
```
coverage report -m
```

To get the coverage in html use the following command.

```
coverage html
```

The html report is normally saved in the /html_cov folder. To access it open html_cov/index.html

The full documentation of coverage is online at https://coverage.readthedocs.io/en/coverage-5.2.1/ but here is an example HTML output that how coverage is displayed.



Coverage report

file:///C:/Users/Robert/Google%20Drive/teaching/2015/FIT4004/example_code/pyunit/htmlcov/index.html

Coverage report: 35%

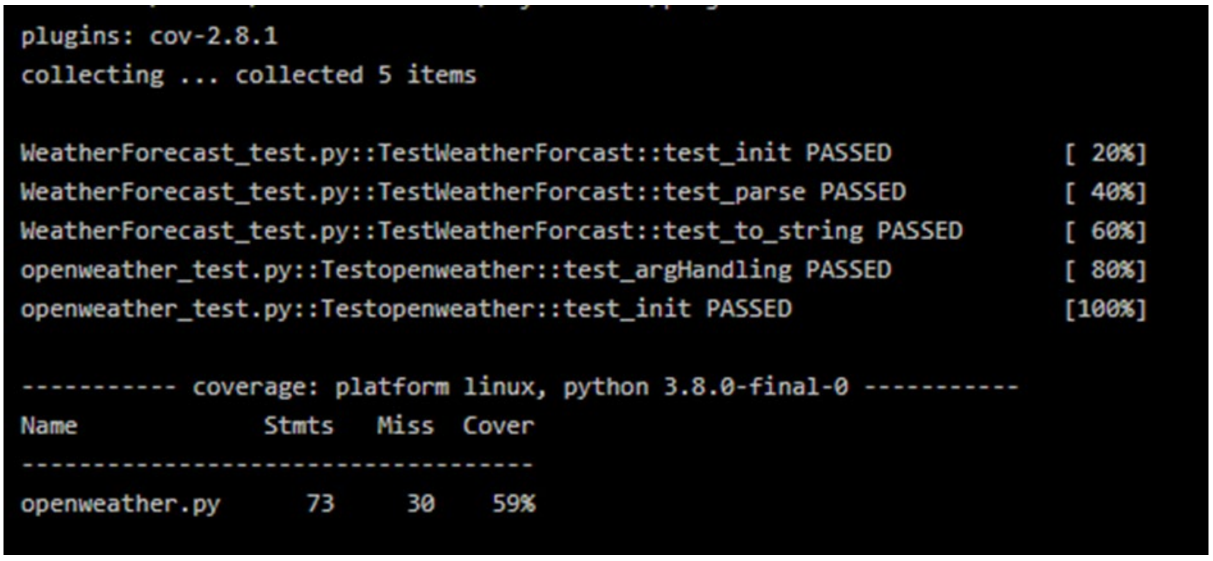
Module	statements	missing	excluded	branches	partial	coverage
/usr/local/lib/python3.4/dist-packages/mock	1249	756	0	554	76	34%
stack	33	11	0	6	1	64%
stacktest	25	0	0	2	1	96%
Total	**1307**	**767**	**0**	**562**	**78**	**35%**

coverage.py v3.7.1

Figure 7: An example of HTML coverage report.

You can further explore the detailed report by clicking the files.

You can also display coverage on command line as well. Below is a sample example of command line usage of coverage



```
plugins: cov-2.8.1
collecting ... collected 5 items

WeatherForecast_test.py::TestWeatherForcast::test_init PASSED            [ 20%]
WeatherForecast_test.py::TestWeatherForcast::test_parse PASSED           [ 40%]
WeatherForecast_test.py::TestWeatherForcast::test_to_string PASSED       [ 60%]
openweather_test.py::Testopenweather::test_argHandling PASSED            [ 80%]
openweather_test.py::Testopenweather::test_init PASSED                   [100%]

----------- coverage: platform linux, python 3.8.0-final-0 -----------
Name              Stmts   Miss  Cover
------------------------------------
openweather.py       73     30    59%
```

Figure 8: An example of command line report of coverage