# Unit Testing

Alexander Dean

April 6, 2020

## 1 Introduction

When programming the notion of testing is used to understand if there are any errors in code. It can catch errors such as syntax errors, assertion errors, name errors, . . . However, we can also test that a process has worked as expected, for example in predictive modelling, we can test that the distribution of predictions matches our prior beliefs. It can be a daunting task to test a whole script, therefore it makes sense to test code in units and this practice is termed unit testing. It is often required that the units are easily identifiable, perhaps the code is written in classes and functions, where each of the units are independent from each other. In a development environment we create a separate tests folder containing separate test scripts for the code that is being tested. Some key terminology similar to [3] is given as:

1. Test step - Define a test to be run (e.g turning the headlights on in a car)
2. Test assertion - What outcomes are we expecting (e.g check if the headlights are on)
3. Test case - Combination of a test step and assertion
4. Test framework - This is a platform that has all the tools and libraries organised and set in a structure to enable automation testing. An analogy can be made to a train station, a train stations set up (platform, ticket counter, trains, ticket office, ...) have a structure that make it very easy to catch a train [1]
5. A test runner - Sets up the execution of tests and provides the outcome to the user.

## 2 Writing unit tests

We will explore testing in python using the unittest module. All unit test scripts need to be saved with filename according to the convention test_*.py, this will enable a clear project structure and running of tests. Lets say we create a script even_check.py that contains a function to test if a number is even. A similar process is given in [2]

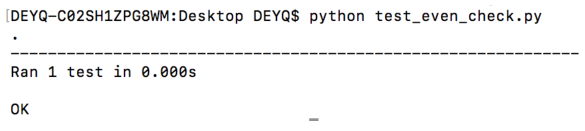
```
#check if a number is even
def is_even(number):
    #is the modulus after dividing by 2 zero?
    if number%2 ==0:
        return 1
    else:
        return 0
```

W now test the is_even function by creating a file test_even_check.py with the following test.

```
#testing the is_even function
import unittest
from even_check import is_even
#create a class that inherits from the TestCase class
class EvensTestCase(unittest.TestCase):
    """Tests for even_check.py"""
    #convert functions to methods by passing self as an argument
    def test_is_2_even(self):
        """Is 2 even?"""
        self.assertTrue(is_even(2))

#run the tests when the script is passed through the intepreter
if __name__ == '__main__':
    unittest.main()
```

This file has created a unit test with a single unit test case; which tests if 2 is even. Of course, since 2 is even, we would expect this test to run successfully and the results of this test are shown below.



```
DEYQ-C02SH1ZPG8WM:Desktop DEYQ$ python test_even_check.py
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

Figure 1: Unit test results

We have made use of the package unittest and pythons built in unit test framework. We can explain the detail contained in the test_even_check.py file. Any member function in a class derived from unittest.TestCase and whose name begins test will be run, when unittest.main() is called. Note when a python interpreter reads a source file it does an important step, it sets variables like__name__. In fact, when the interpreter reads the script, it assigns the hard coded string "__main__" to the __name__ variable. Note an interpreter is an application which will run the python script. So clearly, in our test script we will be invoking the unititest.main() method. This method triggers other methods within unittest, one of which is runTests. So the last few lines of code are instructions to run the tests that we have defined within the script test_even_check.py, when the program is ran through a python interpreter. In our script above we have made use of unittest's assert true method which asserts that the argument passed to it is true. There are a wide range of other asset methods available listed in [4].

## 3 Principles of unit testing

There are some key guidelines for how unit tests should be carried out in the F.I.R.S.T framework as shown in [5].

1. Fast - tests should run quickly
2. Isolated/Independent - The order tests are run should not matter, the assert should test only a single logical outcome
3. Repeatable - A test method should not depend on any data in the instance that it is running, each test should set up or arrange its own data
4. Self-Validating - No manual inspection required to check whether the test has passed or failed
5. Thorough - Should cover most cases

## References

[1] https://www.youtube.com/watch?v=8WKhMsS9HBw. Accessed on 2020-04-06.

[2] https://jeffknupp.com/blog/2013/12/09/improve-your-python-understanding-unit-testing. Accessed on 2020-04-06.

[3] https://realpython.com/python-testing/unit-tests-vs-integration-tests. Accessed on 2020-04-06.

[4] https://docs.python.org/3/library/unittest.html#assert-methods. Accessed on 2020-04-06.

[5] https://github.com/ghsukumar/SFDC_Best_Practices/wiki/F.I.R.S.T-Principles-of-Unit-Testing. Accessed on 2020-04-06.