An Introduction to Testing (and Unittest)

****Note: if you're after just the unittest stuff skip to page 3*

Testing?

A lot of the fun from programming is the creation of the code that does whatever we want it to do, whether that be a solution to a problem we're trying to solve, or something like adding some additional functionality to a game we're making. Regardless of whatever your code is intended to do, you have to ask yourself "how do I know my code is doing what I want it to?"

"Yeah I test all the time, plenty of print statements in my code"

Many of us start answering this question by printing output to a console and manually checking it by eye. It's not a bad place to start, however this becomes quickly unfeasible for large scale projects – sometimes you need to run a lot of tests, over a great period of time, and don't want to spend ages looking at a screen. Not only that, but human error when checking things isn't exactly uncommon. So we need some kind of automated testing, that we can reuse repeatedly, and doesn't require human verification.

Enter Unittest

So there are many ways we can do automated testing, however Python comes with a nice package called Unittest which has many nice features that take the heavy lifting out of testing. There's only one thing we to do:

```
import unittest
```

Let's go through an example to demonstrate how to create some tests with unittest. In this example, we'll make a Calculator class which will perform some basic arithmetic calculations. For simplicity, no documentation or type hints will be added.

Calculator Time

```
class Calculator():

    def __init__(self):
        pass

    def add(self,x,y):
        return x + y

    def subtract(self,x,y):
        return x - y

    def multiply(self,x,y):
        return x * y

    def divide(self,x,y):
        return x / y
```

Here's our Calculator class defined with our basic operations of add, subtract, multiply, and divide. Let's make an assumption that x and y are only going to ever be integers.

Now we need to create some tests to ensure that these methods do what we want them to!

What tests would be good idea?

Actual Unit Testing

It's generally accepted to be a good idea to test each individual component of a piece of software. This idea is known as unit testing. What this usually means is that each method or function that we create should have tests done on it – each one of these tests is called a *test case*. Obviously we cannot make an infinite number of tests to test everything, so the question remains, what tests would be a good idea?

Testing Strategy

There are different approaches to what kinds of test cases to make – the two major ones you'll encounter are Boundary Value Analysis and Equivalence Partitioning.

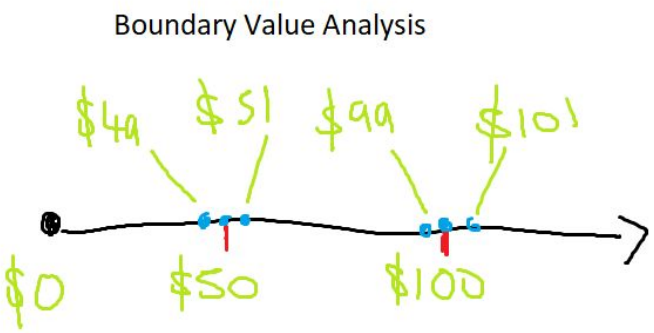
Boundary Value Analysis *(BVA)* involves testing inputs that are around a boundary of some description. It assumes that weird or unwanted behaviour of our functions is going to happen near certain boundaries. For example, say we have a function that will send an alert to a user when their bank balance drops below $100, and also if their bank balance drops below $50 (let's assume the balance is only in positive integers).

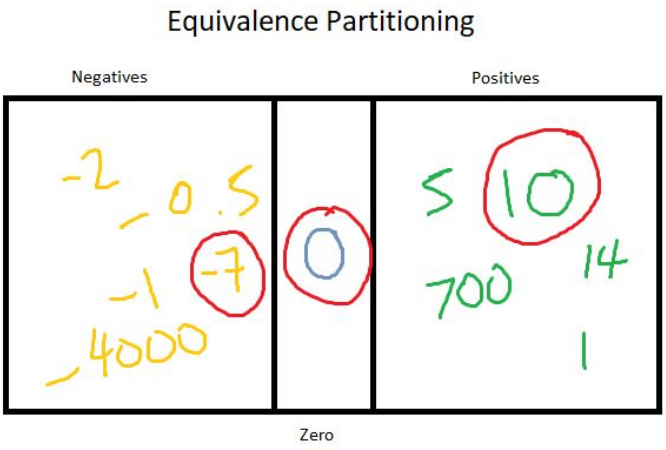
We would want to test $49, $50, $51, $99, $100, $101 as these are the values around the boundary.

Equivalence Partitioning *(EP)* involves partitioning the potential inputs into mutually exclusive sections, and testing one input from each of the sections. It assumes that any input in a given section is (for our purposes) equivalent to any other input in that section, so if a bug or unwanted behaviour would or would not occur for some input in a section, it would or wouldn't occur (respectively) for any input in that section.

For example, for our Calculator class above, we could partition the potential inputs for our multiply method into positives, negatives, and zero. We would then pick one value from each segment to test (randomly selected from these segments if we wanted, but shouldn't matter).

You may notice that these two concepts somewhat overlap – a selection of -1,0,1 for an Equivalence Partition for multiply is also a BVA approach to testing multiply. They are similar, but allow for a different focus depending on where problems are expected to be.





UNITTEST STARTS NOW

As a reminder, here's our Calculator class:

```
class Calculator():

    def __init__(self):
        pass

    def add(self,x,y):
        return x + y

    def subtract(self,x,y):
        return x - y

    def multiply(self,x,y):
        return x * y

    def divide(self,x,y):
        return x / y
```

Let's get started making tests for it!

First up, we need to create a Test Class to use with our Calculator class, and we need to make it inherit from the unittest TestCase class.

```
class TestCalculator(unittest.TestCase):
```

The name here "TestCalculator" is completely arbitrary - call it whatever you want. What isn't arbitrary is the passed class TestCalculator is inheriting from (unittest.TestCase). You need to pass it this for a number of reasons – the biggest being this is how unittest knows TestCalculator is going to be a test class!

Now let's write our first test method (aka test case). Let's write a test method for Calculator's add method. 'calcy' is our pet calculator (aka class instance) we'll use for this.

```
class TestCalculator(unittest.TestCase):

    def test_add(self):
        calcy = Calculator()
        self.assertEqual(calcy.add(2,2), 4)
```

So here we've made "test_add". It's prefixed with "test" specifically because unittest scans your test class for methods beginning with the test prefix – this is how it knows they're test cases. What test_add does is create a Calculator object, then calls a special unittest method called assertEqual on Calculator's add method. assertEqual checks if its parameters are equal, and throws an assertion error if they're not. Here we're saying we want our add method to return 4, and to throw an assertion error otherwise.

Ok, so how do we actually run out test? We need to call unittest's main method. We like to do this in a (hopefully) familiar way:

```
class TestCalculator(unittest.TestCase):

    def test_add(self):
        calcy = Calculator()
        self.assertEqual(calcy.add(2,2), 4)

if __name__ == '__main__':
    unittest.main()
```

The if statement is just making sure we're only doing the unittest.main() call if we're operating straight out of the current module (instead of say, importing this from another file). Refer to the opening documentation in the possible prac solutions file under Week 0 (solution_prac0.py) for more information on this.

So here's our output in the console:

```
.
----------------------------------------------------------------------
Ran 1 test in 0.008s

OK
```

Not a lot huh? All it's really doing is telling us we didn't fail (which isn't to say our method is 100% working, but it's a start...)

Let's try something that's designed to fail – let's make a multiply test:

```
class TestCalculator(unittest.TestCase):

    def test_add(self):
        calcy = Calculator()
        self.assertEqual(calcy.add(2,2), 4)

    def test_multiply(self):
        calcy = Calculator()
        self.assertEqual(calcy.multiply(-1,10), 10)

if __name__ == '__main__':
    unittest.main()
```

Similar situation to the test_add method, only now we've called the multiply method on our Calculator object, and our test is designed to **fail**! Let's see if it does:

```
.F
======================================================================
FAIL: test_multiply (__main__.TestCalculator)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "D:\Calculator unittest.py", line 35, in test_multiply
    self.assertEqual(calcy.multiply(-1,10), 10)
AssertionError: -10 != 10

----------------------------------------------------------------------
Ran 2 tests in 0.008s

FAILED (failures=1)
```

Ok, now here's some output! Never been so happy to see a fail in our lives have we? This output is pretty informative – it tells us what test case failed, which error was raised, which line the case failed on etc. It also tells us that it ran 2 tests but only 1 failed. Pretty nice.

NOTE: As an aside, now seems like a good idea to mention that we've specifically avoided defining an __init__ method for our TestCalculator class – this is because we want to use the __init__ method we inherit from the parent class (unittest.TestCase). Another reason we need to inherit from unittest.TestCase.

It's time to do an example which shows the flexibility and usefulness of doing tests with unittest – let's make a test for the divide method:

```
class TestCalculator(unittest.TestCase):

    def test_add(self):
        calcy = Calculator()
        self.assertEqual(calcy.add(2,2), 4)

    def test_multiply(self):
        calcy = Calculator()
        self.assertEqual(calcy.multiply(-1,10), 10)

    def test_divide(self):
        calcy = Calculator()
        with self.assertRaises(ZeroDivisionError):
            calcy.divide(5,0)


if __name__ == '__main__':
    unittest.main()
```

This test for divide is to test whether our divide method will raise a ZeroDivisionError. We've utilised the with keyword here (not covered here, but Google context managers for more info), which you'll likely encounter in your file I/O adventures. If you don't like 'with', you could do this manually with try and except statements - but this is much neater and nicer. Look at how generous unittest is! Let's have a look at the console after running it:

```
..F
======================================================================
FAIL: test_multiply (__main__.TestCalculator)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "D:\Calculator unittest.py", line 35, in test_multiply
    self.assertEqual(calcy.multiply(-1,10), 10)
AssertionError: -10 != 10

----------------------------------------------------------------------
Ran 3 tests in 0.009s

FAILED (failures=1)
```

Very similar to the previous console output – except you can see we've run 3 tests. Note that despite multiply failing (and raising an assertion error) we still managed to both run and pass the

divide test. This is another very useful aspect to unittest. Let's **fix** the multiply test and **break** the divide test to see what happens:

```
class TestCalculator(unittest.TestCase):

    def test_add(self):
        calcy = Calculator()
        self.assertEqual(calcy.add(2,2), 4)

    def test_multiply(self):
        calcy = Calculator()
        self.assertEqual(calcy.multiply(-1,10), -10)

    def test_divide(self):
        calcy = Calculator()
        with self.assertRaises(ZeroDivisionError):
            calcy.divide(5,1)


if __name__ == '__main__':
    unittest.main()
```

```
.F.
======================================================================
FAIL: test_divide (__main__.TestCalculator)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "D:\Calculator unittest.py", line 40, in test_divide
    calcy.divide(5,1)
AssertionError: ZeroDivisionError not raised

----------------------------------------------------------------------
Ran 3 tests in 0.010s

FAILED (failures=1)
```

Great! Unittest throws an Assertion Error when the ZeroDivisionError we wanted to happen didn't happen. Let's put this back to the original version for now.

The last thing this document will go through is subtests – ways of running different (but very similar) test cases within the same test. Let's say that for whatever reason our Calculator wanted to return an error if subtracting by 0 didn't return a value greater than 0:

```
class TestCalculator(unittest.TestCase):

    def test_add(self):
        calcy = Calculator()
        self.assertEqual(calcy.add(2,2), 4)

    def test_multiply(self):
        calcy = Calculator()
        self.assertEqual(calcy.multiply(-1,10), -10)

    def test_divide(self):
        calcy = Calculator()
        with self.assertRaises(ZeroDivisionError):
            calcy.divide(5,0)

    def test_subtract(self):
        calcy = Calculator()
        for i in range(0,5):
            with self.subTest(i=i):
              self.assertGreater(calcy.subtract(i,1),0)

if __name__ == '__main__':
    unittest.main()
```

We're running a subtest here (again utilising the 'with' keyword) over a number of similar values (from 0 to 4 inclusive), and checking if our subtraction by 1 returns a value greater than 0. The value of i represents which subtest we're in (and which value will be used in the test). Obviously for 0 and 1 we should encounter errors – let's see if we do:

```
...
======================================================================
FAIL: test_subtract (__main__.TestCalculator) (i=0)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "D:\Calculator unittest.py", line 46, in test_subtract
    self.assertGreater(calcy.subtract(i,1),0)
AssertionError: -1 not greater than 0

======================================================================
FAIL: test_subtract (__main__.TestCalculator) (i=1)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "D:\Calculator unittest.py", line 46, in test_subtract
    self.assertGreater(calcy.subtract(i,1),0)
AssertionError: 0 not greater than 0

----------------------------------------------------------------------
Ran 4 tests in 0.010s

FAILED (failures=2)
```

Giving us exactly what we expected. Clearly we can see the benefits here – subtest allows us to do lots of similar tests without instantly returning a fail for the actual test as soon as it encounters an instance within the test that doesn't pass an assertion. In the example above it actually shows the value of i for which the test failed, and all the tests that failed. For reference this is what the output would be if we removed the "with subTest" line:

```
...F
======================================================================
FAIL: test_subtract (__main__.TestCalculator)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "D:\Calculator unittest.py", line 46, in test_subtract
    self.assertGreater(calcy.subtract(i,1),0)
AssertionError: -1 not greater than 0

----------------------------------------------------------------------
Ran 4 tests in 0.013s

FAILED (failures=1)
```

Which is a lot less helpful when trying to debug. Thank goodness for subTest.

This concludes this document's introduction to testing and unittest in Python. However there is much more to learn about testing and the unittest module – for example, check out Python's own documentation on unittest, or go digging through unittest's files if you want a deeper understanding.

Hopefully this is now the beginning of a positive relationship between you and unittest!