

Intro to PyUnit and unit testing

Tutorial 5

Owen Huyn

February 6, 2017

Unit testing quote

“If you don’t like unit testing your product, most likely your customers won’t like to test it either.”

- Anonymous

What is unit testing?

- Unit testing verifies that individual units of code (usually functions) work as intended
- Designed to be **simple**, easy to write and run
- You can test both from a blackbox perspective and a whitebox perspective

Who should write unit tests?

- Developers should test their own code!
- The person who wrote the code usually has the best understanding of what their code does!

So why do we unit test?

- Catches bugs much earlier
- Provides documentation on a specific function
- Helps developer improve the implementation design of a function

Every good developer should be a good tester too!
No one likes to work with someone who doesn't
verify/test if their code works.

What is PyUnit

- PyUnit is a testing framework that belongs to the **xUnit** class of testing frameworks
- Similar to JUnit (Java), CppUnit (C++)
- Knowledge is transferrable to another xUnit framework regardless of language

Alright, I'm sold. How do I get started?

- The PyUnit library already comes preinstalled into Python!
- The library used to write tests is under the 'unittest' module

Demo

Let's get started, I encourage everyone to pull out their laptops and follow along.

Don't be afraid to ask any questions!

Demo

- For our demo, let's test our first assignment!
- The functions that we need to test should be familiar with everyone

Let's create our first unit testing file!

- To start, create a new Python file in the same directory of our file that we want to test
- You can do this from the command line or any text editor of your choice

<input type="checkbox"/> Name	Date modified	Type	Size
 test_circles	2017-02-02 12:23 PM	Python Source File	1 KB

```
MINGW64:/c/Users/huyno/OneDrive/McMaster/2AA4/T5/src
huyno@DESKTOP-75IIJ86 MINGW64 ~/OneDrive/McMaster/2AA4/T5/src
$ echo >> test_circles.py
huyno@DESKTOP-75IIJ86 MINGW64 ~/OneDrive/McMaster/2AA4/T5/src
$
```

Let's create our first test template

- To start with our unit test, follow this template:

```
test_circles.py x [search] [toggle] [menu]
1  import unittest
2
3  class CirclesTests(unittest.TestCase):
4
5
6
7  if __name__ == '__main__':
8      unittest.main()
```

1. Import the unit test library
2. Write a unit testing class with 'unittest.TestCase' as your argument
3. Line 7 and Line 8 helps run the test file itself

OK, let's write our first test

- Let us test something very simple
- Our first test will be testing the xcoord and ycoord getters of our Circle class

Import the module you are testing

Create some tests here; Each function declaration is one test

In this case, we have two tests.

```
test_circles.py x
1 import unittest
2 from CircleADT import *
3
4 class CirclesTests(unittest.TestCase):
5
6     def test_xcoord_are_equal(self):
7         circle = CircleT(1,2,3)
8         self.assertTrue(circle.xcoord() == 1)
9
10    def test_xcoord_are_not_equal(self):
11        circle = CircleT(1,2,3)
12        self.assertFalse(circle.xcoord() != 1)
13
14 if __name__ == '__main__':
15     unittest.main()

CircleADT.py x
7
8 ## @brief An ADT that represents a circle
9 class CircleT:
10
11     ## @brief CircleT constructor
12     # @details Initializes a CircleT object with a cartesian coordin
13     # _____ and a radius
14     # @param x The x coordinate of the circle center
15     # @param y The y coordinate of the circle center
16     # @param r The radius of the circle
17     # @exception ValueError Throws if supplied radius is negative
18     def __init__(self, x, y, r):
19         if(r < 0):
20             raise ValueError("Radius cannot be negative")
21         self.__x = x
22         self.__y = y
23         self.__r = r
24
25     ## @brief Gets the x coordinate of the circle center
26     # @return The x coordinate of the circle center
27     def xcoord(self):
28         return self.__x
29
30     ## @brief Gets the y coordinate of the circle center
31     # @return The y coordinate of the circle center
32     def ycoord(self):
33         return self.__y
```

What on earth is an assertion????

From Wikipedia:

“... a statement that a predicate (Boolean-valued function, a true–false expression) is expected to always be true at that point in the code. If an assertion evaluates to false at run time, an assertion failure results, which typically causes the program to crash, or to throw an assertion exception.”

ELI5:

Basically, if the test passes that Boolean expression, it will continue. Otherwise the test will fail and it will not continue with that test.

Example of an assertion

```
assertTrue(1 == 1)
```

This passes, and the test continues.

```
assertTrue(1 == 2)
```

This Boolean expression is false, and throws an `AssertionException` which causes the test to fail.

```
assertFalse (1 == 2)
```

This Boolean expression is false, but the assertion asserts that the expression is false thus it passes and the test continues.

Running the tests

- Run this command in your command prompt:

```
python -m unittest <name of your test module>
```

In our demo, it is: `python -m unittest test_circles`

- If you have multiple test files, you can run this to run all of them at once

```
python -m unittest discover
```

Module name is not the same as the class name!!

MINGW64:/c/Users/huyno/onedrive/mcmaster/2aa4/t5/src

```
huyno@DESKTOP-75IIJ86 MINGW64 ~/onedrive/mcmaster/2aa4/t5/src
$ python -m unittest test_circles
```

```
..
```

```
-----
Ran 2 tests in 0.001s
```

```
OK
```

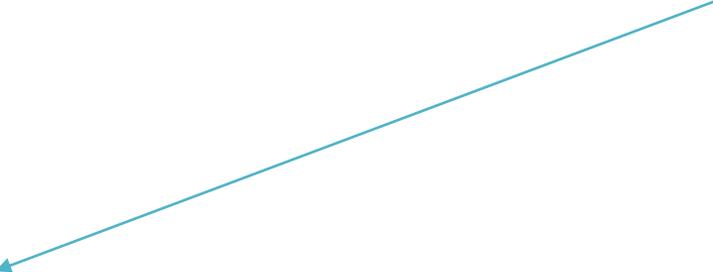
```
huyno@DESKTOP-75IIJ86 MINGW64 ~/onedrive/mcmaster/2aa4/t5/src
$
```

Failed Tests

Let's say we have an example like this:

```
1 class CirclesTests(unittest.TestCase):  
2  
3     def test_xcoord_are_equal(self):  
4         circle = CircleT(1,2,3)  
5         self.assertTrue(circle.xcoord() == 1)  
6  
7     def test_xcoord_are_not_equal(self):  
8         circle = CircleT(1,2,3)  
9         self.assertFalse(circle.xcoord() != 1)  
10  
11     def test_xcoord_will_fail_test(self):  
12         circle = CircleT(1,2,3)  
13         self.assertTrue(circle.xcoord() == 8)
```

This will fail!



When we run our test....

```
huyno@DESKTOP-75IIJ86 MINGW64 ~/onedrive/mcmaster/2aa4/t5/src
$ python -m unittest test_circles
..F
```

```
=====
FAIL: test_xcoord_will_fail_test (test_circles.CirclesTests)
-----
```

```
Traceback (most recent call last):
```

```
  File "test_circles.py", line 16, in test_xcoord_will_fail_test
    self.assertTrue(circle.xcoord() == 8)
```

```
AssertionError: False is not true
```

Where it failed

Reason for failure

```
-----
Ran 3 tests in 0.000s
```

```
FAILED (failures=1)
```

3 total tests, 1 failed, 2 passed

Method	Checks that
<code>assertEqual(a, b)</code>	<code>a == b</code>
<code>assertNotEqual(a, b)</code>	<code>a != b</code>
<code>assertTrue(x)</code>	<code>bool(x) is True</code>
<code>assertFalse(x)</code>	<code>bool(x) is False</code>
<code>assertIs(a, b)</code>	<code>a is b</code>
<code>assertIsNot(a, b)</code>	<code>a is not b</code>
<code>assertIsNone(x)</code>	<code>x is None</code>
<code>assertIsNotNone(x)</code>	<code>x is not None</code>
<code>assertIn(a, b)</code>	<code>a in b</code>
<code>assertNotIn(a, b)</code>	<code>a not in b</code>
<code>assertIsInstance(a, b)</code>	<code>isinstance(a, b)</code>
<code>assertNotIsInstance(a, b)</code>	<code>not isinstance(a, b)</code>

Other assertions

- Documentation can be found here:
<https://docs.python.org/2/library/unittest.html>
- Most of the time, you will use the first 4 assertions, however the other assertions could come in handy
- There are more uncommon assertions (not listed here) that are in the docs

Floating point assertions

```
def assertAlmostEqual(self, first, second, places=None,  
msg=None, delta=None)
```

Checks if the two numbers are equal up until the given number of decimal places (argument: places, default is 7); delta is used as an acceptable range for both numbers

```
def assertNotAlmostEqual(self, first, second,  
places=None, msg=None, delta=None)
```

Similarly here, except this is the logical negation of the first assertion

More into optional arguments here:

http://www.diveintopython.net/power_of_introspection/optional_arguments.html

Redundant code in tests

Let's say we have some redundant code in all our tests:

```
3 class CirclesTests(unittest.TestCase):  
  
3     def test_xcoord_are_equal(self):  
        circle = CircleT(1,2,3)  
        self.assertTrue(circle.xcoord() == 1)  
  
3     def test_xcoord_are_not_equal(self):  
        circle = CircleT(1,2,3)  
        self.assertFalse(circle.xcoord() != 1)
```

This will
run **before**
every test
(setUp)

This will run
after every
test
(tearDown)

```
4 class CirclesTests(unittest.TestCase):
5
6     def setUp(self):
7         self.circle = CircleT(1,2,3)
8
9     def tearDown(self):
10        self.circle = None
11
12        def test_xcoord_are_equal(self):
13            self.assertTrue(circle.xcoord() == 1)
14
15        def test_xcoord_are_not_equal(self):
16            self.assertFalse(circle.xcoord() != 1)
17
18 if __name__ == '__main__':
19     unittest.main()
```

Group Activity:

A more complicated function

- Come up with some tests with your peers for the intersect and insideBox function in the CircleT class
- Try to cover the requirements and edge cases along with any ambiguity

A complete test file

- Refer to `testStatistics.py` for a more complete breakdown on how to test complicated functions

How much should I test?

- Test all requirements in each function
- Cover edge cases that may cause unintended consequences
- Have an acceptable amount of [code coverage](#)

Extra tutorials

PyUnit documentation website:

<https://docs.python.org/2/library/unittest.html>

If you want to really dive into how to be a unit testing pro:

<https://pymotw.com/3/unittest/index.html> (PyUnit tutorial)

Mocking: <http://stackoverflow.com/questions/2665812/what-is-mocking>

Mocking Tutorial: <https://www.toptal.com/python/an-introduction-to-mocking-in-python>