PYTHON TESTING<https://pythontesting.net/>

Python Software Development and Software Testing (posts and podcast)

unittest introduction

January 3, 2013 By Brian<https://pythontesting.net/author/brian-2/>

The unittest test framework is python's xUnit style framework.
It is a standard module that you already have if you've got python version 2.1 or greater.
In this post, I'll cover the basics of how to create and run a simple test using unittest.
Then I'll show how I'm using it to test markdown.py.

Overview of unittest

The unittest module used to be called PyUnit, due to it's legacy as a xUnit style framework.
It works much the same as the other styles of xUnit, and if you're familiar with unit testing in other languages, this framework (or derived versions), may be the most comfortable for you.

The standard workflow is:
1. You define your own class derived from unittest.TestCase.
2. Then you fill it with functions that start with 'test_'.
3. You run the tests by placing `unittest.main()` in your file, usually at the bottom.

One of the many benifits of unittest, that you'll use when your tests get bigger than the toy examples I'm showing on this blog, is the use of 'setUp' and 'tearDown' functions to get your system ready for the tests.

Like the doctest introduction*<https://pythontesting.net/framework/doctest-introduction/>*, I'll run through a simple example first, then show how I'm using unittest for testing markdown.py.

unittest example

Using the same unnecessary_math.py module that I wrote in the doctest intro*<https://pythontesting.net/framework/doctest-introduction/#example>*, here's some example test
code to test my 'multiply' function.

Note:
The module *unnecessary_math* is non-standard and can be found here: implementation of unnecessary_math.py

test_um_unittest.py:

```
import unittest
from unnecessary_math import multiply

class TestUM(unittest.TestCase):

    def setUp(self):
```

```
        pass

    def test_numbers_3_4(self):
        self.assertEqual( multiply(3,4), 12)

    def test_strings_a_3(self):
        self.assertEqual( multiply('a',3), 'aaa')

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

In this example, I've used `assertEqual()`. The unittest framework has a whole bunch of `assertBlah()` style functions like `assertEqual()`. Once you have a reasonable reference*<http://docs.python.org/2/library/unittest.html#unittest.TestCase>* for all of the assert functions bookmarked, working with unnittest is pretty powerful and easy.

Aside from the tests you write, most of what you need to do can be accomplished with the test fixture methods such as setUp, tearDown, setUpClass, tearDownClass, etc.

Running unittests

At the bottom of the test file, we have this code:

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

This allows us to run all of the test code just by running the file.
Running it with no options is the most terse, and running with a '-v' is more verbose, showing which tests
ran.

```
> python test_um_unittest.py
..
----------------------------------------------------------------------
Ran 2 tests in 0.000s

OK
> python test_um_unittest.py -v
test_numbers_3_4 (__main__.TestUM) ... ok
test_strings_a_3 (__main__.TestUM) ... ok

----------------------------------------------------------------------
Ran 2 tests in 0.000s

OK
```

Test discovery

Let's say that you've got a bunch of test files. It would be annoying to have to run each test file separately. That's where test discovery comes in handy.

In our case, all of my test code (one file for now) is in 'simple_example'.
To run all of the unittests in there, use `python -m unittest discover simple_example`, with or without the '-v', like this:

```
> python -m unittest discover simple_example
..
----------------------------------------------------------------------
Ran 2 tests in 0.000s

OK
> python -m unittest discover -v simple_example
test_numbers_3_4 (test_um_unittest.TestUM) ... ok
test_strings_a_3 (test_um_unittest.TestUM) ... ok

----------------------------------------------------------------------
Ran 2 tests in 0.000s

OK
```

unittest example with markdown.py

Now, I'll throw unittest at my markdown.py project.
This is going to be pretty straightforward, as the tests are quite similar to the doctest versions, just formatted with all of the unittest boilerplate stuff, especially since I don't need to make use of startUp or tearDown fixtures.

test_markdown_unittest.py:

```
import unittest
from markdown_adapter import run_markdown

class TestMarkdownPy(unittest.TestCase):

    def setUp(self):
        pass

    def test_non_marked_lines(self):
        '''
        Non-marked lines should only get 'p' tags around all input
        '''
        self.assertEqual(
                run_markdown('this line has no special handling'),
                'this line has no special handling</p>')
```

```
    def test_em(self):
        '''
        Lines surrounded by asterisks should be wrapped in 'em' tags
        '''
        self.assertEqual(
                run_markdown('*this should be wrapped in em tags*'),
                '<p><em>this should be wrapped in em tags</em></p>')

    def test_strong(self):
        '''
        Lines surrounded by double asterisks should be wrapped in 'strong' tags
        '''
        self.assertEqual(
                run_markdown('**this should be wrapped in strong tags**'),
                '<p><strong>this should be wrapped in strong tags</strong></p>')

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

Testing markdown.py

And now we can see that everything is failing (as expected).

```
> python test_markdown_unittest.py
FFF
======================================================================
FAIL: test_em (__main__.TestMarkdownPy)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test_markdown_unittest.py", line 29, in test_em
    '<em>this should be wrapped in em tags</em></p>')
AssertionError: '*this should be wrapped in em tags*' != '<p><em>this should be wrappe

======================================================================
FAIL: test_non_marked_lines (__main__.TestMarkdownPy)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test_markdown_unittest.py", line 21, in test_non_marked_lines
    '<p>this line has no special handling</p>')
AssertionError: 'this line has no special handling' != '<p>this line has no special ha

======================================================================
FAIL: test_strong (__main__.TestMarkdownPy)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test_markdown_unittest.py", line 37, in test_strong
    '<p><strong>this should be wrapped in strong tags</strong></p>')
AssertionError: '**this should be wrapped in strong tags**' != '<p><strong>this shoul

----------------------------------------------------------------------
Ran 3 tests in 0.142s

FAILED (failures=3)
```

One interesting thing to note as compared to doctest. Only actual tests are counted.
I have 3 tests. And unittest gets that right.
Doctest lists 4 tests, with one of them passing. What's the 4th? It's the import statement.
Every statement is counted in doctest, so the counts are quite a bit wacky, if you ask me.
The counts are way more meaningful in unittest.

More unittest info

The python.org page on unittest*<http://docs.python.org/2/library/unittest.html>* is a great source for information on unittest.
If you've got another favorite tutorial or reference for unittest, please leave a comment.

Also, the code shown here is available on github.com/variedthoughts/markdown.py*<https://github.com/variedthoughts/markdown.py>*

Next

Now that the basics of doctest and unittest are done, I'll get into some of the real fun by exploring nose, then py.test.
Then, before getting onto some other fun topics, I probably should get my markdown.py script to do something.
In the process of doing that, I'll probably have at least one post talking about my use of regular expressions in python.

Feedback

I've received some feedback through email and other means.
Thank you to everyone that participates in this discussion.

If you are not comfortable leaving comments on a post, you can use the contact form.
If the contact form doesn't work for you, or you just don't like contact forms, you can email me directly.

The email I have set up for this blog is
brian AT python testing DOT net (no spaces, of course)

I have a handful of ideas for future blog posts, so I cannot promise that I will cover all of the suggestions that people give me, but I will try to fit them in, especially if I can learn something in the process.

Thanks for reading. Keep in touch!

Related posts:

1. **nose introduction** ***(nose introduction)***
2. **pytest introduction** ***(pytest introduction)***
3. **doctest introduction** ***(doctest introduction)***
4. **unittest fixture syntax and flow reference** ***(unittest fixture syntax and flow reference)***
5. **What happens when unittest fixtures fail** ***(What happens when unittest fixtures fail)***

Filed Under: unittest<*https://pythontesting.net/category/framework/unittest/*>
Tagged With: frameworks<*https://pythontesting.net/on/frameworks/*>, pyUnit<*https://pythontesting.net/on/pyunit/*>, unittest<*https://pythontesting.net/on/unittest/*>, xUnit<*https://pythontesting.net/on/xunit/*>

Python Testing with unittest, nose, pytest.
Get up to speed fast on pytest, unittest, and nose.
All in the comfort of your own e-reader.