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

Python Software Development and Software Testing (posts and podcast)

doctest introduction

December 11, 2012 By Brian<https://pythontesting.net/author/brian-2/>

The doctest test framework is a python module that comes prepackaged with Python. This post covers the basics of how to put doctests in your code, and outside of your code, in a separate file. Then I'll show how I'm using it to test markdown.py.

conceptual model of python doctest

This is from python.org<*http://docs.python.org/2/library/doctest.html#module-doctest*>:

> ” The doctest module searches for pieces of text that look like interactive
> Python sessions,
> and then executes those sessions to verify that they work exactly as shown.
> …
>
> “

I like to think of doctest as if I’m actually sitting at a python interactive prompt, and typing stuff in. The doctest is a script that says “My session should look exactly like this. If it doesn’t something is wrong.”

Actually, I think some people do use it that way. They write some module, and then demonstrate how it works in an interactive shell, and copy/paste the session into a docstring as their doctests.

However, that doesn’t work so well in TDD, where I’ve not got the code working before I write the test. With TDD, I’ve really got to think about the exact output of something **before** it works.

When using doctest and TDD, it can end up getting rather iterative:

1. Write some doctests
2. Run the doctests to see that they fail
3. Write some code that should make it pass
4. If it still fails, examine the failure.
5. If it is a false failure, and the doctest is just being too picky, then modify the doctest, possibly
   with doctest flags, then go to 2.
6. If it is a real failure, fix the code, then go to 2.

I have found that some of the nitpicky aspects of doctest can be minimized with the use of an api adapter*<https://pythontesting.net/strategy/software-api-cli-interface-adapters>*. I’ll be using an adapter in the markdown.py example in this post.

doctest example

Here is a simple module with one function in it, along with two doctests embedded in the docstring.

unnecessary_math.py:

```
'''
Module showing how doctests can be included with source code
Each '>>>' line is run as if in a python shell, and counts as a test.
The next line, if not '>>>' is the expected output of the previous line.
If anything doesn't match exactly (including trailing spaces), the test fails.
'''

def multiply(a, b):
    """
    >>> multiply(4, 3)
    12
    >>> multiply('a', 3)
    'aaa'
    """
    return a * b
```

running doctest

You run doctest like this:

```
> python -m doctest <file>
or
> python -m doctest -v <file>
```

The '-v' means verbose. Verbose is real handy when testing your doctests, since doctest doesn't output anything if all of the tests pass.

```
> python -m doctest unnecessary_math.py
> python -m doctest -v unnecessary_math.py
Trying:
    multiply(4, 3)
Expecting:
    12
ok
Trying:
    multiply('a', 3)
Expecting:
    'aaa'
ok
1 items had no tests:
    unnecessary_math
1 items passed all tests:
   2 tests in unnecessary_math.multiply
2 tests in 2 items.
```

```
2 passed and 0 failed.
Test passed.
>
```

You can see in the first run that nothing prints out, since all tests pass.

doctests in a separate file from the code

One of the really cool features of doctest is the ability to put your doctests in a text file. This is especially useful for functional testing, since that allows you to use doctest to test even non-python interfaces.

For our simple math example, I can just put the same code from the docstring into a text file.

test_unnecessary_math.txt:

```
This is a doctest based regression suite for unnecessary_math.py
Each '>>' line is run as if in a python shell, and counts as a test.
The next line, if not '>>' is the expected output of the previous line.
If anything doesn't match exactly (including trailing spaces), the test fails.

>>> from unnecessary_math import multiply
>>> multiply(3, 4)
12
>>> multiply('a', 3)
'aaa'
```

running doctests in separate file

Running doctest on a file is the same as running it on a module.

```
> python -m doctest test_unnecessary_math.txt
> python -m doctest -v test_unnecessary_math.txt
Trying:
    from unnecessary_math import multiply
Expecting nothing
ok
Trying:
    multiply(3, 4)
Expecting:
    12
ok
Trying:
    multiply('a', 3)
Expecting:
    'aaa'
```

```
ok
1 items passed all tests:
   3 tests in test_unnecessary_math.txt
3 tests in 1 items.
3 passed and 0 failed.
Test passed.
```

example with markdown.py

For markdown.py, I don't want to include doctests in the code. Since I'm only testing the external CLI (through an adapter), I will be using the 'doctests in a text file' method.

I'm not going to write tests for the entire syntax right away. My first three tests will be for paragraphs, single asterisk em tags, and double asterisk strong tags.

test_markdown_doctest.txt:

```
To run: python -m doctest test_markdown_doctest.txt
    or: python -m doctest -v test_markdown_doctest.txt

>>> from markdown_adapter import run_markdown

>>> run_markdown('paragraph wrapping')
'<p>paragraph wrapping</p>'

>>> run_markdown('*em tags*')
'<p><em>em tags</em></p>'

>>> run_markdown('**strong tags**')
'<p><strong>strong tags</strong></p>'
```

Well, that's simple enough. I've imported 'run_markdown' from my api adapter. Then I throw some example strings into the script and show what I expect to come out.

testing markdown.py

Here's the output of running doctest on my text file.

```
> python -m doctest test_markdown_doctest.txt
**********************************************************************
File "test_markdown_doctest.txt", line 6, in test_markdown_doctest.txt
Failed example:
    run_markdown('paragraph wrapping')
Expected:
    '<p>paragraph wrapping</p>'
Got:
    'paragraph wrapping'
```

```
**********************************************************************
File "test_markdown_doctest.txt", line 9, in test_markdown_doctest.txt
Failed example:
    run_markdown('*em tags*')
Expected:
    '<p><em>em tags</em></p>'
Got:
    '*em tags*'
**********************************************************************
File "test_markdown_doctest.txt", line 12, in test_markdown_doctest.txt
Failed example:
    run_markdown('**strong tags**')
Expected:
    '<p><strong>strong tags</strong></p>'
Got:
    '**strong tags**'
**********************************************************************
1 items had failures:
   3 of   4 in test_markdown_doctest.txt
***Test Failed*** 3 failures.
```

And with verbose.

```
> python -m doctest -v test_markdown_doctest.txt
Trying:
    from markdown_adapter import run_markdown
Expecting nothing
ok
Trying:
    run_markdown('paragraph wrapping')
Expecting:
    '<p>paragraph wrapping</p>'
**********************************************************************
File "test_markdown_doctest.txt", line 6, in test_markdown_doctest.txt
Failed example:
    run_markdown('paragraph wrapping')
Expected:
    '<p>paragraph wrapping</p>'
Got:
    'paragraph wrapping'
Trying:
    run_markdown('*em tags*')
Expecting:
    '<p><em>em tags</em></p>'
**********************************************************************
File "test_markdown_doctest.txt", line 9, in test_markdown_doctest.txt
Failed example:
    run_markdown('*em tags*')
Expected:
    '<p><em>em tags</em></p>'
Got:
    '*em tags*'
Trying:
    run_markdown('**strong tags**')
Expecting:
    '<p><strong>strong tags</strong></p>'
**********************************************************************
File "test_markdown_doctest.txt", line 12, in test_markdown_doctest.txt
Failed example:
    run_markdown('**strong tags**')
```

```
Expected:
    '<p><strong>strong tags</strong></p>'
Got:
    '**strong tags**'
**********************************************************************
1 items had failures:
   3 of   4 in test_markdown_doctest.txt
4 tests in 1 items.
1 passed and 3 failed.
***Test Failed*** 3 failures.
```

As you can see. Once you've convinced yourself that your tests are correct, the verbose setting doesn't add much. You will get plenty of output without verbose if there are errors.

In my case, everything FAILED!!!. But that's good, because I haven't implemented anything real yet, I just have a stub*<https://pythontesting.net/markdown/stub-markdown/>*.

more doctest info

All of the examples in this post are available in the github markdown.py project*<https://github.com/variedthoughts/markdown.py>*.The math example is in a folder called 'simple_doctest_example'.

The python.org site has pretty good information about using doctest*<http://docs.python.org/2/library/doctest.html>*.
On that same page is the writeup on how to use text files for your doctests*<http://docs.python.org/2/library/doctest.html#simple-usage-checking-examples-in-a-text-file>*.

Doug Hellmann has a great writeup on doctest that I highly recommend.It's called Testing through documentation*<http://www.doughellmann.com/PyMOTW/doctest/>* and it covers many of the problems that you may run into including dealing with multiple lines, whitespace, unpredictable output, etc.

I will cover some of these aspects as I get further into the implementation and testing of markdown.py*<https://github.com/variedthoughts/markdown.py>*.

next

Next up, I’ll take a look at implementing the same tests using unittest, also sometimes referred to as PyUnit.

Related posts:

1. **pytest introduction** ***(pytest introduction)***
2. **unittest introduction** ***(unittest introduction)***
3. **nose introduction** ***(nose introduction)***
4. **Stub for markdown.py** ***(Stub for markdown.py)***
5. **regex search and replace example scripts** ***(regex search and replace example scripts)***

Filed Under: doctest*<https://pythontesting.net/category/framework/doctest/>*
Tagged With: doctest*<https://pythontesting.net/on/doctest/>*, markdown*<https://pythontesting.net/on/markdown-2/>*