Python Testing

with unittest, nose, pytest

Efficient and effective testing
using the 3 top python testing frameworks

By Brian Okken

Python Testing with unittest, nose, pytest

Efficient and effective testing using the 3 top python testing frameworks

Brian Okken

This book is for sale at http://leanpub.com/pythontesting

This version was published on 2014-09-19

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Contents

Introduction

PythonTesting.net[1] is a blog where I write about software testing.

The most popular posts so far are the tutorials on how to get started using the 3 most popular testing frameworks:

- unittest
- nose
- pytest

I've organized a bunch of the most poplar posts into a sequence that I think makes sense, and pulled them into this book format.

I did this becuase I personally would have liked this stuff in an eBook format when I started learning about testing with Python. So, I hope it's helpful for you.

How I got started writing about this stuff.

I've been a professional software developer since 1996.

I've been using Python for about the last 10 years. Mostly for testing.

One day I needed a test framework to run a bunch of pre-existing test scripts that weren't set up to use any franework. Because of Pythons introspection facilities, it seemed like it would be pretty easy to implement.

I knew somewhat about unittest. But I really didn't want to derive classes just to get test functions running.

The first incantation was pretty simple, and was implemented quickly.

However, the more we used it, the more new features we needed. So it grew.

Finally, one day, I looked on the web to see what test frameworks were available.

I found out more about unittest. And I found nose.
And then pytest.

But the documentation (for all of them) was confusing. At least to me.

I decided to find out for myself, and explore all three.

[1] http://pythontesting.net

At first I thought I would discover that one was far superior to all the rest.
But I don't think that's the case.

I do think pytest is by far the coolest and most advanced.
However, unittest keeps improving, and is no slouch.
And although nose isn't really in development as far as I can tell, lots of people still use it successfully.

Along the way I've learned lots about all three, and also some on doctest.

That's where pythonTesting.net[2] comes in. I am trying to share all that I know (and can clearly speak about) regarding these frameworks.

This book is a fund raiser

The proceeds from this book do a few things.

1. Encourage me to keep writing content on pythontesting.net[3].
2. Help offset the cost of putting this content and more into a professionally published book on the subject. See next section.
3. Help pay off the laptop I just bought so I can write more.
4. Let my family know that this is a good thing for me to be spending my time on.

[2] http://pythontesting.net
[3] http://pythontesting.net

unittest

unittest introduction

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 work flow 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 benefits 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.

I'll run through a simple example first, then show how I'm using unittest for testing markdown.py.

unittest example

Here is a simple module called unnecessary_math.py.

unnecessary_math.py:

```
1  def multiply(a, b):
2      return a * b
```

Here's some example test code to test my 'multiply' function.

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[4] 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.

[4] http://docs.python.org/2/library/unittest.html#unittest.TestCase

```
> 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'),
                '<p>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
    '<p><em>this should be wrapped in em tags</em></p>')
AssertionError: '*this should be wrapped in em tags*' != '<p><em>this should be \
wrapped in em tags</em></p>'

======================================================================
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 spec\
ial handling</p>'

======================================================================
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 \
should be wrapped in strong tags</strong></p>'

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

FAILED (failures=3)
```

More unittest info

The python.org page on unittest[5] is a great source for information on unittest.

[5] http://docs.python.org/2/library/unittest.html

unittest fixtures

Software Test Fixtures

The term **test fixtures** really means two things.
Test fixtures are the resources and initial conditions that a test needs to operate correctly and independently from other tests.
The phrase has also grown to mean the functions and methods that are used to do that resource and environment handling.

For the rest of this post, I'm really referring to the functions and methods when I say **fixtures**. Test fixtures are methods and functions that run before and after a test. The intent is to provide developers hooks to set up preconditions needed for the test, and cleanup after the test. In many cases, this will be allocating, opening, or connecting to some resource in the setUp, and deallocating, closing, or disconnecting in the tearDown. However, that's just the intent. You can use these really however you want to use them.

One great use for fixtures is to set up structures or variables the same way for all tests.
This is to make sure that tests can run individually as well as a set and in any order.

Common Case Example

The most common fixture methods are setUp and tearDown.
The deserve to be the most common, because they are the ones that allow test independence.
The `setUp()` method runs before every test.
The `tearDown()` method runs after every test.

Here's some code.

```
1  import unittest
2
3  class TestLists(unittest.TestCase):
4
5      def setUp(self):
6          print('')
7          print('in %s - setUp()' % self.id())
8          self.myList = [1, 2, 3, 4]
9
10     def test_len(self):
```

```
11          print('in %s - test_len()' % self.id())
12          self.assertEqual( len(self.myList), 4 )
13          self.myList.append(-1)
14          self.assertEqual( len(self.myList), 5 )
15
16      def test_min(self):
17          print('in %s - test_min()' % self.id())
18          self.assertEqual( min(self.myList) , 1 )
19
20      def tearDown(self):
21          print('in %s - tearDown()' % self.id()
```

And here's the output.

```
1  $ python -m unittest unittest_fixtures_1
2
3  in unittest_fixtures_1.TestLists.test_len - setUp()
4  in unittest_fixtures_1.TestLists.test_len - test_len()
5  in unittest_fixtures_1.TestLists.test_len - tearDown()
6  .
7  in unittest_fixtures_1.TestLists.test_min - setUp()
8  in unittest_fixtures_1.TestLists.test_min - test_min()
9  in unittest_fixtures_1.TestLists.test_min - tearDown()
10 .
11 ----------------------------------------------------------------------
12 Ran 2 tests in 0.000s
13
14 OK
```

Note that the tests are wrapped with `setUp()` and `tearDown()` just as promised.

What's that self.id()?

It is good to note I'm using the `id()` method that is part of `unittest.TestCase` to get the name of the current test. This is valid during the test method, as well as setUp, tearDown, and any methods called from the test method.

Full Test Fixture Example

Although `setUp()` and `tearDown()` are the methods that allow us to make sure each test can run independently and in any order, we have other methods available as well. I think this is a complete list.

- `setUp()` / tearDown()‘ - before and after test methods
- `setUpClass()` / `tearDownClass()` - before and after a class of tests
- `setUpModule()` / `tearDownModule()` - before and after a module of tests
- Cleanup functions - extra tearDown methods that can be added at runtime to any test method during setUp, or during the test method itself.

Here’s some code with everything but cleanup functions.

```
1  import unittest
2
3  def setUpModule():
4      'called once, before anything else in this module'
5      print('in module %s - setupModule() ' % __name__)
6
7  def tearDownModule():
8      'called once, after everything else in this module'
9      print('in module %s - tearDownModule() ' % __name__)
10
11 class TestFixtures(unittest.TestCase):
12
13     @classmethod
14     def setUpClass(cls):
15         'called once, before any tests'
16         print('in class %s - setUpClass() ' % cls.__name__)
17
18     @classmethod
19     def tearDownClass(cls):
20         'called once, after all tests, if setUpClass successful'
21         print('in class %s - tearDownClass() ' % cls.__name__)
22
23     def setUp(self):
24         'called multiple times, before every test method'
25         print('in setUp()')
26
27     def tearDown(self):
28         'called multiple times, after every test method'
29         print('in tearDown()')
30
31     def test_1(self):
32         'a test'
33         print('in test_1()')
34
```

```
    def test_2(self):
        'another test'
        print('in test_2()')
```

Full Test Fixture Flow

```
$ python -m unittest unittest_fixtures.TestFixtures
in module unittest_fixtures - setupModule()
in class TestFixtures - setUpClass()
in setUp()
in test_1()
in tearDown()
.in setUp()
in test_2()
in tearDown()
.in class TestFixtures - tearDownClass()
in module unittest_fixtures - tearDownModule()

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

OK
```

Adding Cleanup Calls

Extra *cleanup* methods can be added from either a test or a setUp method.
Cleanup functions are called AFTER tearDown() but BEFORE tearDownClass()

```
class TestAddCleanup(TestFixtures):

    def setUp(self):
        TestFixtures.setUp(self)
        # --- add a cleanup method fixture for all tests
        def cleanup_a():
            print('in cleanup_a()')
        self.addCleanup(cleanup_a)

    def test_3(self):
        # --- add a cleanup method fixture for just this test
        def cleanup_b():
            print('in cleanup_b()')
```

```
14          self.addCleanup(cleanup_b)
15          print('in test_3()')
```

Output

```
1  $ python -m unittest unittest_fixtures.TestAddCleanup
2  in module unittest_fixtures - setupModule()
3  in class TestAddCleanup - setUpClass()
4  in setUp()
5  in test_1()
6  in tearDown()
7  in cleanup_a()
8  .in setUp()
9  in test_2()
10 in tearDown()
11 in cleanup_a()
12 .in setUp()
13 in test_3()
14 in tearDown()
15 in cleanup_b()
16 in cleanup_a()
17 .in class TestAddCleanup - tearDownClass()
18 in module unittest_fixtures - tearDownModule()
19
20 ----------------------------------------------------------------------
21 Ran 3 tests in 0.000s
22
23 OK
```

Skipping tests within setUp()

In the setUp method, you can decide to skip a test.
If skipped, the test will not be run.
ALSO, the tearDown method will not be run.

```
class TestSkip(TestFixtures):
    def setUp(self):
        TestFixtures.setUp(self)
        currentTest = self.id().split('.')[-1]
        if currentTest == 'test_2':
            self.skipTest('reason for skipping')
            # the 'reason' will displayed if '-v/--verbose' flag used
```

Output

```
$ python -m unittest -q unittest_fixtures.TestSkip
in module unittest_fixtures - setupModule()
in class TestSkip - setUpClass()
in setUp()
in test_1()
in tearDown()
in setUp()
in class TestSkip - tearDownClass()
in module unittest_fixtures - tearDownModule()
----------------------------------------------------------------------
Ran 2 tests in 0.000s

OK (skipped=1)
```

nose

nose introduction

This post has several examples, and covers fixtures, test discovery, asserts, running options, and running unittests and doctests.

Nose's tagline is "nose extends unittest to make testing easier".
It's is a fairly well known python unit test framework, and can run doctests, unittests, and "no boilerplate" tests.

It is a good candidate for a go-to test framework.

I think a smart developer should get familiar doctest[6], unittest[7], pytest[8], and nose. Then decide if one of those makes the most sense for them, or if they want to keep looking for features only found in other frameworks.
That's of course the reason why I'm writing this series. So I guess that last bit goes without saying.

No boilerplate, some api

A basic test file for nose is pretty simple, without any boilerplate code, without required classes to drive from, without unnecessary imports, and without any extra api.

```
1  from unnecessary_math import multiply
2
3  def test_numbers_3_4():
4      assert multiply(3,4) == 12
```

This is identical to the simple test shown in my pytest intro[9].
There are differences between how you have to write your tests for the two frameworks once you get into extra features of the frameworks, like fixtures, plugins, assert mechanisms, etc.
I'm going to leave a full comparison of pytest and nose to a future post.

Why do I say 'some api'? Well, when you get into fixtures (like setup/teardown, etc), there is some nose api that is needed in the tests. I'll get into that in the fixture section.

---

[6] http://pythontesting.net/framework/doctest-introduction/
[7] http://pythontesting.net/framework/unittest-introduction/
[8] http://pythontesting.net/framework/pytest-introduction/
[9] http://pythontesting.net/framework/pytest-introduction/#no_boilerplate

Nose example

For completeness in following the styles of previous framework introductions, here is the full basic test.
This only differs from above that I've added another test function.

```
from unnecessary_math import multiply

def test_numbers_3_4():
    assert multiply(3,4) == 12

def test_strings_a_3():
    assert multiply('a',3) == 'aaa'
```

Running nose

To run nose, use the *nosetests* command that comes with nose.

```
nosetests test_um_nose.py
```

And with verbose:

```
nosetests -v test_um_nose.py
```

Here's an example run both with and without verbose:

```
> nosetests test_um_nose.py
..
----------------------------------------------------------------------
Ran 2 tests in 0.000s

OK
> nosetests -v test_um_nose.py
simple_example.test_um_nose.test_numbers_3_4 ... ok
simple_example.test_um_nose.test_strings_a_3 ... ok

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

OK
```

Nose fixtures

Nose extends the unittest fixture model of setup/teardown. We can add specific code to run:

- at the beginning and end of a module of test code (setup_module/teardown_module)
  To get this to work, you just have to use the right naming rules.
- at the beginning and end of a class of test methods (setup_class/teardown_class) To get this to work, you have to use the right naming rules, and include the '@classmethod' decorator.
- before and after a test function call (setup_function/teardown_function) You can use any name. You have to apply them with the '@with_setup' decorator imported from nose.
  You can also use direct assignment, which I'll show in the example.
- before and after a test method call (setup/teardown) To get this to work, you have to use the right name.

The easiest fixtures to add are:

- **setup_module()** function: runs before anything else in the file
- **teardown_module()** function: runs after everything else in the file

And if you use a class to define some tests:

- **setup()** method: runs before every test method
- **teardown()** method: runs after every test method

You can also set non-class based test functions to have setup/teardown functions, but you have to import the 'with_setup' decorator from nose, like so:

from nose import with_setup # optional

```
def my_setup_function():
    pass

def my_teardown_function():
    pass

@with_setup(my_setup_function, my_teardown_function)
def test_numbers_3_4():
    assert multiply(3,4) == 12
```

If you don't like to use decorators, you can also assign the setup and teardown attributes like this:

```
test_numbers_3_4.setup = my_setup_function
test_numbers_3_4.teardown = my_teardown_function
```

However, I think that's a bit awkward.

With classes, you can set a setup/teardown for the class, but you do it differently.
You need to make sure the methods are class methods using the 'classmethod' decorator, and name them correctly, like so:

```
class TestUM:

    @classmethod
    def setup_class(cls):
        print ("setup_class() before any methods in this class")

    @classmethod
    def teardown_class(cls):
        print ("teardown_class() after any methods in this class")
```

It works, it's just that you have to keep the syntax straight for all the different rules for different fixtures. Here they are all together.

```
from nose import with_setup # optional

from unnecessary_math import multiply

def setup_module(module):
    print ("") # this is to get a newline after the dots
    print ("setup_module before anything in this file")

def teardown_module(module):
    print ("teardown_module after everything in this file")

def my_setup_function():
    print ("my_setup_function")

def my_teardown_function():
    print ("my_teardown_function")

@with_setup(my_setup_function, my_teardown_function)
def test_numbers_3_4():
    print 'test_numbers_3_4  <============================ actual test code'
```

```
21      assert multiply(3,4) == 12
22
23  @with_setup(my_setup_function, my_teardown_function)
24  def test_strings_a_3():
25      print 'test_strings_a_3  <============================ actual test code'
26      assert multiply('a',3) == 'aaa'
27
28
29  class TestUM:
30
31      def setup(self):
32          print ("TestUM:setup() before each test method")
33
34      def teardown(self):
35          print ("TestUM:teardown() after each test method")
36
37      @classmethod
38      def setup_class(cls):
39          print ("setup_class() before any methods in this class")
40
41      @classmethod
42      def teardown_class(cls):
43          print ("teardown_class() after any methods in this class")
44
45      def test_numbers_5_6(self):
46          print 'test_numbers_5_6()  <============================ actual test cod\
47  e'
48          assert multiply(5,6) == 30
49
50      def test_strings_b_2(self):
51          print 'test_strings_b_2()  <============================ actual test cod\
52  e'
53          assert multiply('b',2) == 'bb'
```

To see it in action, I'll use the **-s** option, which turns off output capture.
This will show the order of the different fixture calls.

```
> nosetests -s test_um_nose_fixtures.py
....
setup_module before anything in this file
setup_class() before any methods in this class
TestUM:setup() before each test method
test_numbers_5_6()  <============================ actual test code
TestUM:teardown() after each test method
TestUM:setup() before each test method
test_strings_b_2()  <============================ actual test code
TestUM:teardown() after each test method
teardown_class() after any methods in this class
my_setup_function
test_numbers_3_4  <============================ actual test code
my_teardown_function
my_setup_function
test_strings_a_3  <============================ actual test code
my_teardown_function
teardown_module after everything in this file

----------------------------------------------------------------------
Ran 4 tests in 0.001s

OK
```

Testing markdown.py

This is also identical to code that can be run from py.test[10].
It's similar to unittest code, but without boilerplate, and with simple assert calls instead of assertEquals.
Again, I'm using the API adapter[11] to cleanly call markdown functionality.

Here's the code to use nose to test markdown.py:

[10] http://pythontesting.net/framework/pytest-introduction/#testing_markdown
[11] http://pythontesting.net/strategy/software-api-cli-interface-adapters/

```
from markdown_adapter import run_markdown

def test_non_marked_lines():
    print ('in test_non_marked_lines')
    assert run_markdown('this line has no special handling') \
            == '<p>this line has no special handling</p>'

def test_em():
    print ('in test_em')
    assert run_markdown('*this should be wrapped in em tags*') \
            == '<p><em>this should be wrapped in em tags</em></p>'

def test_strong():
    print ('in test_strong')
    assert run_markdown('**this should be wrapped in strong tags**') \
            == '<p><strong>this should be wrapped in strong tags</strong></p>'
```

And here's the output:

```
> nosetests  test_markdown_nose.py
FFF
======================================================================
FAIL: test_markdown_nose.test_non_marked_lines
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\python27\lib\site-packages\nose-1.2.1-py2.7.egg\nose\case.py", line 1\
97, in runTest
    self.test(*self.arg)
  File "E:\python_notes\repo\markdown.py-dev\test_markdown_nose.py", line 13, in\
 test_non_marked_lines
    == '<p>this line has no special handling</p>'
AssertionError:
-------------------- >> begin captured stdout << ---------------------
in test_non_marked_lines

--------------------- >> end captured stdout << ----------------------

======================================================================
FAIL: test_markdown_nose.test_em
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\python27\lib\site-packages\nose-1.2.1-py2.7.egg\nose\case.py", line 1\
```

```
97, in runTest
    self.test(*self.arg)
  File "E:\python_notes\repo\markdown.py-dev\test_markdown_nose.py", line 18, in\
 test_em
    == '<p><em>this should be wrapped in em tags</em></p>'
AssertionError:
-------------------- >> begin captured stdout << ---------------------
in test_em

--------------------- >> end captured stdout << ----------------------

======================================================================
FAIL: test_markdown_nose.test_strong
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\python27\lib\site-packages\nose-1.2.1-py2.7.egg\nose\case.py", line 1\
97, in runTest
    self.test(*self.arg)
  File "E:\python_notes\repo\markdown.py-dev\test_markdown_nose.py", line 23, in\
 test_strong
    == '<p><strong>this should be wrapped in strong tags</strong></p>'
AssertionError:
-------------------- >> begin captured stdout << ---------------------
in test_strong

--------------------- >> end captured stdout << ----------------------

----------------------------------------------------------------------
Ran 3 tests in 0.137s

FAILED (failures=3)
```

All of the tests are failing.
Although the line numbers of the failures, along with the test function names, are printed, it's not real obvious from the report what's wrong.

Nose assert_equals

If we are using lots of **assert something == somethingElse** type tests, and we are committed to using nose for testing, we can use nose tools to make the report a bit obvious about what the failure is. I'm going to rewrite the tests from above using **nose.tools.assert_equals**:

```
from nose.tools import assert_equals
from markdown_adapter import run_markdown

def test_non_marked_lines():
    print ('in test_non_marked_lines')
    assert_equals(run_markdown('this line has no special handling'),
        '<p>this line has no special handling</p>')

def test_em():
    print ('in test_em')
    assert_equals( run_markdown('*this should be wrapped in em tags*'),
        '<p><em>this should be wrapped in em tags</em></p>')

def test_strong():
    print ('in test_strong')
    assert_equals( run_markdown('**this should be wrapped in strong tags**'),
        '<p><strong>this should be wrapped in strong tags</strong></p>')
```

Nose's assert_equals works a lot like unittest's assertEquals. Now lets look at the output:

```
> nosetests test_markdown_nose_assert_equals.py
FFF
======================================================================
FAIL: test_markdown_nose_assert_equals.test_non_marked_lines
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\python27\lib\site-packages\nose-1.2.1-py2.7.egg\nose\case.py", line 1\
97, in runTest
    self.test(*self.arg)
  File "E:\python_notes\repo\markdown.py-dev\test_markdown_nose_assert_equals.py\
", line 14, 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 spec\
ial handling</p>'
-------------------- >> begin captured stdout << ---------------------
in test_non_marked_lines

--------------------- >> end captured stdout << ----------------------

======================================================================
FAIL: test_markdown_nose_assert_equals.test_em
----------------------------------------------------------------------
```

```
Traceback (most recent call last):
  File "C:\python27\lib\site-packages\nose-1.2.1-py2.7.egg\nose\case.py", line 1\
97, in runTest
    self.test(*self.arg)
  File "E:\python_notes\repo\markdown.py-dev\test_markdown_nose_assert_equals.py\
", line 19, in test_em
    '<p><em>this should be wrapped in em tags</em></p>')
AssertionError: '*this should be wrapped in em tags*' != '<p><em>this should be \
wrapped in em tags</em></p>'
-------------------- >> begin captured stdout << ---------------------
in test_em

--------------------- >> end captured stdout << ----------------------

======================================================================
FAIL: test_markdown_nose_assert_equals.test_strong
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\python27\lib\site-packages\nose-1.2.1-py2.7.egg\nose\case.py", line 1\
97, in runTest
    self.test(*self.arg)
  File "E:\python_notes\repo\markdown.py-dev\test_markdown_nose_assert_equals.py\
", line 24, 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 \
should be wrapped in strong tags</strong></p>'
-------------------- >> begin captured stdout << ---------------------
in test_strong

--------------------- >> end captured stdout << ----------------------

----------------------------------------------------------------------
Ran 3 tests in 0.139s

FAILED (failures=3)
```

Now the output makes it more obvious what's wrong.

Test discovery

I use the same naming conventions for nose as I do for py.test.

- Name my test modules/files starting with 'test_'.
- Name my test functions starting with 'test_'.
- Name my test classes starting with 'Test'.
- Name my test methods starting with 'test_'.
- Make sure all packages with test code have an '**init**.py' file.

These rules work just fine for me. This isn't the complete list of rules. If you want to do something different, look at the nose documentation for finding tests[12]

Running unittests from nose

Nose finds and runs unittests with no problem, and with no extra steps.
Here I'll run the tests from the unittest intro[13]:

```
1  > nosetests test_um_unittest.py
2  ..
3  ----------------------------------------------------------------------
4  Ran 2 tests in 0.000s
5
6  OK
7  > nosetests -v test_um_unittest.py
8  test_numbers_3_4 (simple_example.test_um_unittest.TestUM) ... ok
9  test_strings_a_3 (simple_example.test_um_unittest.TestUM) ... ok
10
11 ----------------------------------------------------------------------
12 Ran 2 tests in 0.001s
13
14 OK
```

Running doctests from nose

Nose can run doctests, supposedly[14]. However, I couldn't get it to work on doctests in a separate file[15] method, using **test_unnecessary_math.txt**.

I tried several of the options, with no luck.
If you know what I'm doing wrong, please let me know.

[12] https://nose.readthedocs.org/en/latest/finding_tests.html
[13] http://pythontesting.net/framework/unittest-introduction/
[14] https://nose.readthedocs.org/en/latest/plugins/doctests.html#module-nose.plugins.doctests
[15] http://pythontesting.net/framework/doctest-introduction/#separate_file

More nose info (links)

- nose.readthedocs[16] - nose official documentation (I think)
- pypi[17] - download links and multiple versions
- nose-dev[18] - google group on nose

Examples on github

All of the examples here are available in the markdown.py project[19] on github.

[16] https://nose.readthedocs.org/en/latest
[17] http://pypi.python.org/pypi/nose
[18] https://groups.google.com/forum/?fromgroups#!forum/nose-dev
[19] https://github.com/okken/markdown.py

nose support for unittest style fixtures

I ran nosetests on the tests written for unittest from my unittest fixture[20] post.
No surprises.
Nose supports unittest style fixtures:

- module: setUpModule()/tearDownModule()
- class: setUpClass()/tearDownClass()
- around methods: setUp()/tearDown()
- add cleanup functions: addCleanup() called from setUp() and from a test
- skipping tests dynamically: testSkip() called from setUp()
- error conditions: don't run the test or the matching tearDown if setUp fails. This is true for module, class, and method fixtures.

Here's the output. (error cases omitted).

```
> nosetests -q -s test_fixtures:TestFixtures
in module test_fixtures - setUpModule()
in class TestFixtures - setUpClass()
in test_1 - setUp()
in test_1 - test_1()
in test_1 - tearDown()
in test_2 - setUp()
in test_2 - test_2()
in test_2 - tearDown()
in class TestFixtures - tearDownClass()
in module test_fixtures - tearDownModule()
----------------------------------------------------------------------
Ran 2 tests in 0.071s

OK

> nosetests -q -s test_fixtures:TestAddCleanup
in module test_fixtures - setUpModule()
```

[20] http://pythontesting.net/framework/unittest/unittest-fixtures/

```
in class TestAddCleanup - setUpClass()
in test_1 - setUp()
in test_1 - test_1()
in test_1 - tearDown()
in test_1 - cleanup_b()
in test_1 - cleanup_a()
in test_2 - setUp()
in test_2 - test_1()
in test_2 - tearDown()
in test_2 - cleanup_a()
in class TestAddCleanup - tearDownClass()
in module test_fixtures - tearDownModule()
----------------------------------------------------------------------
Ran 2 tests in 0.079s

OK

> nosetests -q -s test_fixtures:TestSkip
in module test_fixtures - setUpModule()
in class TestSkip - setUpClass()
in test_1 - setUp()
in test_1 - test_1()
in test_1 - tearDown()
in test_2 - setUp()
in class TestSkip - tearDownClass()
in module test_fixtures - tearDownModule()
----------------------------------------------------------------------
Ran 2 tests in 0.067s

OK (SKIP=1)
```

See the post unittest fixture syntax and flow reference[21] for the source and a discussion of unittest style fixtures.

I'm not posting the output for the error cases[22], mostly because it's long and boring.
However, I did verify that the proper control flow in failure cases is upheld when unittest tests are run with nose.

[21] http://pythontesting.net/framework/unittest/unittest-fixtures/
[22] http://pythontesting.net/framework/unittest/when-unittest-fixtures-fail/

nose fixture reference

I'm going to cover nose setup and teardown fixtures at the package, module, class, method, and function level.
This isn't about what code to put into the fixtures, just about the syntax and flow.
And a bit about naming conventions.

Although I talked about the fixtures in the nose introduction[23], I think this post will work better as a reference.

Method vs. function

In the discussion below, I'm distinguishing a difference between *method* and *function*.
I'm following a convention from the python documentation[24]:

```
A method is a function that "belongs to" an object.
```

In other words, if a function is NOT in a class, it's a *function*.
If it IS in a class, it's a *method*.

Package level fixtures

Add 'setup_package()' and 'teardown_package()' functions to the '**init**.py' of a package.
'setup_package' will run before any test modules within the package.
'teardown_package' will run after all of the modules are run (if setup succeeded).

```
def setup_package():
    pass

def teardown_package():
    pass
```

Module level fixtures

Module fixtures bracket the execution of everthing in the module (test classes and test functions).
These functions only run once each.

[23] http://pythontesting.net/framework/nose/nose-introduction/#fixtures
[24] http://docs.python.org/2/tutorial/classes.html#instance-objects

```
def setup_module():
    pass

def teardown_module():
    pass
```

Class level fixtures

Class fixtures bracket the execution of everthing in a class.
These functions only run once each.

```
class TestClass():

    @classmethod
    def setup_class(cls):
        pass

    @classmethod
    def teardown_class(cls):
        pass
```

Class method level fixtures

Class method fixtures bracket the execution of each and every test method in the class.
These functions run multiple times, once for each test method.

```
class TestClass():

    def setup(self):
        pass

    def teardown(self):
        pass
```

Function level fixtures

Function level fixtures are the oddball of the bunch.
Nose doesn't do any naming convention detection for this.
It's all done with the '@with_setup' decorator, which has to be included from 'nose.tools'.

```
from nose.tools import with_setup

def setup_function():
    pass

def teardown_function():
    pass

@with_setup(setup_function, teardown_function)
def test_something():
    pass
```

This does allow you to have different fixtures for different tests.

```
def foo():
    pass

def bar():
    pass

@with_setup(foo, bar)
def test_something_else():
    pass
```

However, please choose names that make sense.
It also allows you to forget to add the decorator, so be careful.

```
@with_setup(setup_function, teardown_function)
def test_a():
    pass

@with_setup(setup_function, teardown_function)
def test_b():
    pass

def test_c():  # Does this lack a fixture on purpose? or did someone forget?
    pass
```

Full example

__init__.py

```
from __future__ import print_function

def setup_package():
    print('')
    print(__name__, '__init__.py : setup_package() ============================\
===========')

def teardown_package():
    print(__name__, '__init__.py : teardown_package() =========================\
===========')
```

test_using_classes.py

```
from __future__ import print_function

def setup_module():
    print(__name__, ': setup_module() ~~~~~~~~~~~~~~~~~~~~~~')

def teardown_module():
    print(__name__, ': teardown_module() ~~~~~~~~~~~~~~~~~~')


class TestClass():

    @classmethod
    def setup_class(cls):
        print(__name__, ': TestClass.setup_class() ----------')

    @classmethod
    def teardown_class(cls):
        print(__name__, ': TestClass.teardown_class() -------')

    def setup(self):
        print(__name__, ': TestClass.setup()  - - - - - - - -')

    def teardown(self):
        print(__name__, ': TestClass.teardown() - - - - - - -')

    def test_method_1(self):
        print(__name__, ': TestClass.test_method_1()')

    def test_method_2(self):
        print(__name__, ': TestClass.test_method_2()')
```

test_using_functions.py

```
from __future__ import print_function
from nose.tools import with_setup

def setup_module():
    print(__name__, ': setup_module() ~~~~~~~~~~~~~~~~~~~~~~')

def teardown_module():
    print(__name__, ': teardown_module() ~~~~~~~~~~~~~~~~~~')

def setup_function():
    "attached with 'with_setup'decorator"
    print(__name__, ': setup_function() - - - - - - - - -')

def teardown_function():
    "attached with 'with_setup'decorator"
    print(__name__, ': teardown_function()  - - - - - - -')

def test_func_1():
    print(__name__, ': test_func_1()')

def test_func_2():
    print(__name__, ': test_func_2()')

@with_setup(setup_function, teardown_function)
def test_func_3():
    print(__name__, ': test_func_3()')
```

Control flow

I put the files in a `testNoseFixtures` directory. Running the tests with `-s` so I can see the output, produces the following.

```
> nosetests -s testNosePackage
.....
testNosePackage __init__.py : setup_package() ================================\
======
testNosePackage.test_using_classes : setup_module() ~~~~~~~~~~~~~~~~~~~~~~
testNosePackage.test_using_classes : TestClass.setup_class() ----------
testNosePackage.test_using_classes : TestClass.setup()  - - - - - - - - -
testNosePackage.test_using_classes : TestClass.test_method_1()
testNosePackage.test_using_classes : TestClass.teardown() - - - - - - - -
testNosePackage.test_using_classes : TestClass.setup()  - - - - - - - - -
testNosePackage.test_using_classes : TestClass.test_method_2()
testNosePackage.test_using_classes : TestClass.teardown() - - - - - - - -
testNosePackage.test_using_classes : TestClass.teardown_class() -------
testNosePackage.test_using_classes : teardown_module() ~~~~~~~~~~~~~~~~~~
testNosePackage.test_using_functions : setup_module() ~~~~~~~~~~~~~~~~~~~~~~
testNosePackage.test_using_functions : test_func_1()
testNosePackage.test_using_functions : test_func_2()
testNosePackage.test_using_functions : setup_function() - - - - - - - - - -
testNosePackage.test_using_functions : test_func_3()
testNosePackage.test_using_functions : teardown_function()  - - - - - - - -
testNosePackage.test_using_functions : teardown_module() ~~~~~~~~~~~~~~~~~~
testNosePackage __init__.py : teardown_package() ==============================\
======

----------------------------------------------------------------------
Ran 5 tests in 0.001s

OK
```

Alternative names

Nose is pretty forgiving about naming conventions for fixtures.
I'll list the alternative names for the different fixtures.
However, I strongly encourage you to use the names listed above.
The names listed above are, my opinion, easiest to read.
The exception to this is setup_function/teardown_function, since those are possibly custom for every test function, use whatever you like.

setup_package
: setup, setUp, or setUpPackage teardown_package

: teardown, tearDown, or tearDownPackage setup_module

setup, setUp, or setUpModule teardown_module

teardown, tearDown, or tearDownModule setup_class

setupClass, setUpClass, setupAll, or setUpAll teardown_class

teardownClass, tearDownClass, teardownAll, or tearDownAll. setup (class method fixtures)

setUp teardown (class method fixtures)

tearDown setup_function / teardown_function

can be named anything, since it's attached to a function with '@with_setup'

pytest

pytest introduction

I think of pytest as the run-anything, no boilerplate, no required api, use-this-unless-you-have-a-reason-not-to test framework.
This is really where testing gets fun.
As with previous intro's on this site, I'll run through an overview, then a simple example, then throw pytest at my markdown.py project. I'll also cover fixtures, test discovery, and running unittests with pytest.

No boilerplate, no required api

The doctest[25] and unittest[26] both come with Python.
They are pretty powerful on their own, and I think you should at least know about those frameworks, and learn how to run them at least on some toy examples, as it gives you a mental framework to view other test frameworks.

With unittest, you a very basic test file might look like this:

```
1  import unittest
2  from unnecessary_math import multiply
3
4  class TestUM(unittest.TestCase):
5
6      def test_numbers_3_4(self):
7          self.assertEqual( multiply(3,4), 12)
```

The style of deriving from `unittest.TestCase` is something unittest shares with it's xUnit counterparts like JUnit. I don't want to get into the history of xUnit style frameworks. However, it's informative to know that inheritance is quite important in some languages to get the test framework to work right. But this is Python. We have very powerful introspection and runtime capabilities, and very little information hiding. Pytest takes advantage of this. An identical test as above could look like this if we remove the boilerplate:

[25] http://pythontesting.net/framework/doctest-introduction/
[26] http://pythontesting.net/framework/unittest-introduction/

```
from unnecessary_math import multiply

def test_numbers_3_4():
    assert( multiply(3,4) == 12 )
```

Yep, three lines of code. (Four, if you include the blank line.)
There is no need to `import unnittest`.
There is no need to derive from `TestCase`.
There is no need to for special `self.assertEqual()`, since we can use Python's built in `assert` statement.

This works in pytest. Once you start writing tests like this, you won't want to go back.

However, you may have a bunch of tests already written for doctest or unittest.
Pytest can be used to run doctests and unittests.
It also claims to support some twisted trial tests (although I haven't tried this).

You can extend pytest using plugins you pull from the web, or write yourself.
I'm not going to cover plugins in this article, but I'm sure I'll get into it in a future article.

You will sometimes see pytest referred to as py.test.
I use this convention:
pytest : the project py.test : the command line tool that runs pytest I'm not sure if that's 100% accurate according to how the folks at pytest.org use the terms.

pytest example

Using the same unnecessary_math.py module that I wrote in the doctest intro[27], this is some example test code to test the 'multiply' function.

```
from unnecessary_math import multiply

def test_numbers_3_4():
    assert multiply(3,4) == 12

def test_strings_a_3():
    assert multiply('a',3) == 'aaa'
```

Running pytest

To run pytest, the following two calls are identical:

[27] http://pythontesting.net/framework/doctest-introduction/#example

```
python -m pytest test_um_pytest.py
py.test test_um_pytest.py
```

And with verbose:

```
python -m pytest -v test_um_pytest.py
py.test -v test_um_pytest.py
```

I'll use `py.test`, as it's shorter to type. Here's an example run both with and without verbose: ‘

```
> py.test test_um_pytest.py
============================= test session starts ==============================
platform win32 -- Python 2.7.3 -- pytest-2.2.4
collecting ... collected 2 items

test_um_pytest.py ..

========================== 2 passed in 0.05 seconds ===========================


> py.test -v test_um_pytest.py
============================= test session starts ==============================
platform win32 -- Python 2.7.3 -- pytest-2.2.4 -- C:\python27\python.exe
collecting ... collected 2 items

test_um_pytest.py:12: test_numbers_3_4 PASSED
test_um_pytest.py:15: test_strings_a_3 PASSED

========================== 2 passed in 0.02 seconds ===========================
```

‘

pytest fixtures

Although unittest does allow us to have setup and teardown, pytest extends this quite a bit. We can add specific code to run:

- at the beginning and end of a module of test code (setup_module/teardown_module)
- at the beginning and end of a class of test methods (setup_class/teardown_class)
- alternate style of the class level fixtures (setup/teardown)

- before and after a test function call (setup_function/teardown_function)
- before and after a test method call (setup_method/teardown_method)

We can also use pytest style fixtures, which are covered in pytest fixtures nuts and bolts[28].

I've modified our simple test code with some fixture calls, and added some print statements so that we can see what's going on.
Here's the code:
‘

```
1  from unnecessary_math import multiply
2
3  def setup_module(module):
4      print ("setup_module      module:%s" % module.__name__)
5
6  def teardown_module(module):
7      print ("teardown_module   module:%s" % module.__name__)
8
9  def setup_function(function):
10     print ("setup_function    function:%s" % function.__name__)
11
12 def teardown_function(function):
13     print ("teardown_function function:%s" % function.__name__)
14
15 def test_numbers_3_4():
16     print 'test_numbers_3_4  <============================ actual test code'
17     assert multiply(3,4) == 12
18
19 def test_strings_a_3():
20     print 'test_strings_a_3  <============================ actual test code'
21     assert multiply('a',3) == 'aaa'
22
23
24 class TestUM:
25
26     def setup(self):
27         print ("setup             class:TestStuff")
28
29     def teardown(self):
30         print ("teardown          class:TestStuff")
31
```

[28] http://pythontesting.net/framework/pytest/pytest-fixtures-nuts-bolts/

```
32      def setup_class(cls):
33          print ("setup_class       class:%s" % cls.__name__)
34
35      def teardown_class(cls):
36          print ("teardown_class    class:%s" % cls.__name__)
37
38      def setup_method(self, method):
39          print ("setup_method      method:%s" % method.__name__)
40
41      def teardown_method(self, method):
42          print ("teardown_method   method:%s" % method.__name__)
43
44      def test_numbers_5_6(self):
45          print 'test_numbers_5_6  <============================ actual test code'
46          assert multiply(5,6) == 30
47
48      def test_strings_b_2(self):
49          print 'test_strings_b_2  <============================ actual test code'
50          assert multiply('b',2) == 'bb'
```

‘

To see it in action, I’ll use the **-s** option, which turns off output capture.
This will show the order of the different fixture calls.

‘

```
1  > py.test -s test_um_pytest_fixtures.py
2  ============================= test session starts ==============================
3  platform win32 -- Python 2.7.3 -- pytest-2.2.4
4  collecting ... collected 4 items
5
6  test_um_pytest_fixtures.py ....
7
8  ========================== 4 passed in 0.07 seconds ===========================
9  setup_module      module:test_um_pytest_fixtures
10 setup_function    function:test_numbers_3_4
11 test_numbers_3_4  <============================ actual test code
12 teardown_function function:test_numbers_3_4
13 setup_function    function:test_strings_a_3
14 test_strings_a_3  <============================ actual test code
15 teardown_function function:test_strings_a_3
16 setup_class       class:TestUM
```

```
setup_method      method:test_numbers_5_6
setup             class:TestStuff
test_numbers_5_6  <========================== actual test code
teardown          class:TestStuff
teardown_method   method:test_numbers_5_6
setup_method      method:test_strings_b_2
setup             class:TestStuff
test_strings_b_2  <========================== actual test code
teardown          class:TestStuff
teardown_method   method:test_strings_b_2
teardown_class    class:TestUM
teardown_module   module:test_um_pytest_fixtures
```

‘

Testing markdown.py

The test code to test markdown.py is going to look a lot like the unittest version[29], but without the boilerplate.

I’m also using an API adapter[30] introduced in a previous post.

Here’s the code to use pytest to test markdown.py:

```
from markdown_adapter import run_markdown

def test_non_marked_lines():
    print ('in test_non_marked_lines')
    assert run_markdown('this line has no special handling') == \
            '<p>this line has no special handling</p>'

def test_em():
    print ('in test_em')
    assert run_markdown('*this should be wrapped in em tags*') == \
            '<p><em>this should be wrapped in em tags</em></p>'

def test_strong():
    print ('in test_strong')
    assert run_markdown('**this should be wrapped in strong tags**') == \
            '<p><strong>this should be wrapped in strong tags</strong></p>'
```

And here’s the output:

[29] http://pythontesting.net/framework/unittest-introduction/#example_markdown

[30] http://pythontesting.net/strategy/software-api-cli-interface-adapters/

```
> py.test test_markdown_pytest.py
============================= test session starts ==============================
platform win32 -- Python 2.7.3 -- pytest-2.2.4
collecting ... collected 3 items

test_markdown_pytest.py F.F

=================================== FAILURES ===================================
_____________________________ test_non_marked_lines _____________________________

    def test_non_marked_lines():
        print ('in test_non_marked_lines')
>       assert run_markdown('this line has no special handling') ==
                '<p>this line has no special handling</p>'
E       assert 'this line ha...cial handling' == '<p>this line ... handling</p>'
E         - this line has no special handling
E         + this line has no special handling
E         ? +++                                  ++++

test_markdown_pytest.py:14: AssertionError
------------------------------- Captured stdout ------------------------------
in test_non_marked_lines
_____________________________________ test_strong _____________________________________

    def test_strong():
        print ('in test_strong')
>       assert run_markdown('**this should be wrapped in strong tags**') ==
                '<p><strong>this should be wrapped in strong tags</strong></p>'
E       assert '**this shoul...strong tags**' == '<p><strong>th...</strong></p>'
E         - **this should be wrapped in strong tags**
E         + <strong>this should be wrapped in strong tags</strong>

test_markdown_pytest.py:24: AssertionError
------------------------------- Captured stdout ------------------------------
in test_strong
====================== 2 failed, 1 passed in 0.30 seconds ======================\

```

You'll notice that all of them are failing. This is on purpose, since I haven't implemented any real markdown code yet.
However, the formatting of the output is quite nice.
It's quite easy to see why the test is failing.

Test discovery

The unittest module comes with a 'discovery' option.
Discovery is just built in to pytest.
Test discovery was used in my examples to find tests within a specified module.
However, pytest can find tests residing in multiple modules, and multiple packages, and even find unittests and doctests.
To be honest, I haven't memorized the discovery rules.
I just try to do this, and at seems to work nicely:

- Name my test modules/files starting with 'test_'.
- Name my test functions starting with 'test_'.
- Name my test classes starting with 'Test'.
- Name my test methods starting with 'test_'.
- Make sure all packages with test code have an '**init**.py' file.

If I do all of that, pytest seems to find all my code nicely.
If you are doing something else, and are having trouble getting pytest to see your test code, then take a look at the pytest discovery documentation[31].

Running unittests from pytest

To show how pytest handles unittests, here's a sample run of pytest on the simple unittests I wrote in the unittest introduction[32]: '

```
1  > py.test test_um_unittest.py
2  ============================= test session starts ==============================
3  platform win32 -- Python 2.7.3 -- pytest-2.2.4
4  collecting ... collected 2 items
5
6  test_um_unittest.py ..
7
8  ========================== 2 passed in 0.07 seconds ===========================
9  > py.test -v test_um_unittest.py
10 ============================= test session starts ==============================
11 platform win32 -- Python 2.7.3 -- pytest-2.2.4 -- C:\python27\python.exe
12 collecting ... collected 2 items
13
```

[31] http://pytest.org/latest/example/pythoncollection.html
[32] http://pythontesting.net/framework/unittest-introduction/

```
test_um_unittest.py:15: TestUM.test_numbers_3_4 PASSED
test_um_unittest.py:18: TestUM.test_strings_a_3 PASSED

========================== 2 passed in 0.06 seconds ==========================
```

As you can see, I didn't provide any extra options, pytest finds unittests automatically.

Running doctests from pytest

You can run some doctests from pytest, according to the documentation.
However, with my examples of putting doctests in text files[33], I can't figure out a way to get pytest to run them.

I've tried several attempts, and keep getting into import error problems: ‘

```
> py.test --doctest-modules test_unnecessary_math.txt
============================= test session starts ==============================
platform win32 -- Python 2.7.3 -- pytest-2.2.4
collecting ... collected 1 items

test_unnecessary_math.txt F

=================================== FAILURES ===================================
_____________________________________ [doctest] _____________________________________
001 This is a doctest based regression suite for unnecessary_math.py
002 Each '>>>' line is run as if in a python shell, and counts as a test.
003 The next line, if not '>>>' is the expected output of the previous line.
004 If anything doesn't match exactly (including trailing spaces), the test fail\
s.
005
006 >>> from unnecessary_math import multiply
UNEXPECTED EXCEPTION: ImportError('No module named unnecessary_math',)
Traceback (most recent call last):

  File "C:\python27\lib\doctest.py", line 1289, in __run
    compileflags, 1) in test.globs

  File "", line 1, in

ImportError: No module named unnecessary_math
```

[33] http://pythontesting.net/framework/doctest-introduction/

```
26
27 E:\python_notes\repo\markdown.py-dev\simple_example\test_unnecessary_math.txt:6:\
28  UnexpectedException
29 ========================== 1 failed in 0.06 seconds ==========================
```

Note: Commenters on the blog did point out the problems and how to fix it. In future editions, I'll re-run this and figure out what went wrong and how to fix it

pytest fixtures

Dealing with fixtures is one of the areas where pytest really shines. This is rather an incredible understatement.

The xunit style of test fixtures that is used in both unittest and nose is of course supported with with pytest. And pytest rocks at this. But there is another way to deal with fixtures. It's to think of fixtures as a set of resources that need to be set up before a test starts, and cleaned up after. Test functions, methods, classes, name wich fixtures they need. This way, fixtures aren't set up for tests that don't need them.

This focus on the fixture as a modular resource, possibly set up once for many tests, or perhaps for each test, is a really cool way to treat fixtures.
However, it took me a while to really get my head around it.

I've been thinking about it for months, and have started many, many attempts at a 'pytest fixture' post.

Truth. I don't think one post will do it justice.
So, I'll take a bit at a time, and try to cover it in a way that I can express how cool I think pytest fixtures are, and hopefly completely enough that you can use them effectively.

It's not complicated to use. It's really quite simple. But it's different enough from what I was used to that I think a series of posts would be best.

pytest full support of unittest fixtures

You can run unittest tests from pytest.

The fixtures run the same as they would with unittest.

pytest xUnit style fixtures

I'm going to cover the syntax for pytest support for xUnit style fixtures.
Then I'll give a more reasonable and typical example, using just one set of fixture functions.
And then address the issue of having tests mixed in a file. Some that need the resource, and some that don't.

Depending on what scope you want your fixtures, you define setup/teardown pairs.

- Module (setup_module/teardown_module)
  - Sets things up once for the module, teardown after everything.
- Function (setup_function/teardown_function)
  - Wraps every test function with calls.
  - Gets called multiple times, once for each function
- Class (setup_class/teardown_class)
  - Like module level, but for classes, once for a class.
- Method (setup_method/teardown_method)
  - Like function level, but for classes.
  - Gets called multiple times, once for each test method in a class

Example using module, function, class, and method fixtures

```
from __future__ import print_function

def setup_module(module):
    print('\nsetup_module()')

def teardown_module(module):
    print('teardown_module()')

def setup_function(function):
    print('\nsetup_function()')

def teardown_function(function):
    print('\nteardown_function()')

def test_1():
```

```
16      print('-  test_1()')
17
18  def test_2():
19      print('-  test_2()')
20
21
22  class TestClass:
23
24      @classmethod
25      def setup_class(cls):
26          print ('\nsetup_class()')
27
28      @classmethod
29      def teardown_class(cls):
30          print ('teardown_class()')
31
32      def setup_method(self, method):
33          print ('\nsetup_method()')
34
35      def teardown_method(self, method):
36          print ('\nteardown_method()')
37
38      def test_3(self):
39          print('- test_3()')
40
41      def test_4(self):
42          print('- test_4()')
```

And lets run it to see the flow. I did remove some of the extra blank lines for this post.

```
1  $ py.test -s -v test_xunit_style_fixtures.py
2  ============================== test session starts ============
3  platform darwin -- Python 2.7.5 -- pytest-2.3.4 -- /usr/bin/python
4  collected 4 items
5
6  test_xunit_style_fixtures.py:17: test_1
7  setup_module()
8  setup_function()
9  -  test_1()
10 PASSED
11 teardown_function()
12
```

```
13 test_xunit_style_fixtures.py:20: test_2
14 setup_function()
15 -  test_2()
16 PASSED
17 teardown_function()
18
19 test_xunit_style_fixtures.py:40: TestClass.test_3
20 setup_class()
21 setup_method()
22 - test_3()
23 PASSED
24 teardown_method()
25
26 test_xunit_style_fixtures.py:43: TestClass.test_4
27 setup_method()
28 - test_4()
29 PASSED
30 teardown_method()
31 teardown_class()
32 teardown_module()
33
34
35 ============================== 4 passed in 0.01 seconds =========
```

Realistic example

Typically, you don't throw all of the fixture types together.
Most of the time, one style is enough, depending what you are setting up, initializing, etc. and if it needs re-initialized before every test, and cleaned up after every test.

So, lets make things a bit simpler.

I've got a resource, called resource_a. I know, boring name. It's just something that needs a setup and a teardown function.

This really could be any sort of resource:

- temp file
- temp directory
- database connection
- db transaction that needs rolled back after testing
- open socket connection
- a signal generator putting out a test signal

- you get the drift

In this example, I've used method level fixtures so the setup/teardown happens at the beginning and end of the module, once for all the tests. Maybe it's an expensive operation or something.

Here's our simpler example with a resource.

```
from __future__ import print_function

def resource_a_setup():
    print('resources_a_setup()')

def resource_a_teardown():
    print('resources_a_teardown()')

def setup_module(module):
    print('\nsetup_module()')
    resource_a_setup()

def teardown_module(module):
    print('\nteardown_module()')
    resource_a_teardown()

def test_1_that_needs_resource_a():
    print('test_1_that_needs_resource_a()')
```

And the output.

```
$ py.test -s -v test_realistic_one_func.py
================================ test session starts ================
platform darwin -- Python 2.7.5 -- pytest-2.3.4 -- /usr/bin/python
collected 1 items

test_realistic_one_func.py:17: test_1_that_needs_resource_a
setup_module()
resources_a_setup()
test_1_that_needs_resource_a()
PASSED
teardown_module()
resources_a_teardown()

============================ 1 passed in 0.01 seconds =============
```

Adding another test function

Then we add a test that actually doesn't need the resource: ‘

```
# ...
def test_2_that_does_not():
    print('\ntest_2_that_does_not()')
```

And we re-run it.

```
$ py.test -s -v test_realistic_two_funcs.py
================================ test session starts ============================\
=====
platform darwin -- Python 2.7.5 -- pytest-2.3.4 -- /usr/bin/python
collected 2 items

test_realistic_two_funcs.py:17: test_1_that_needs_resource_a
setup_module()
resources_a_setup()
test_1_that_needs_resource_a()
PASSED
test_realistic_two_funcs.py:20: test_2_that_does_not
test_2_that_does_not()
PASSED
teardown_module()
resources_a_teardown()

============================== 2 passed in 0.01 seconds =========================\
=====
```

This isn't really a problem so far.
Since the first test needs the resource, it's fine the way we are doing things.

Problem: the resource is set up even when we don't need it.

If we just want to run one function, the second one, that doesn't need the resource, the fixture is run anyway.

```
$ py.test -s -v test_realistic_two_funcs.py::test_2_that_does_not
============================== test session starts ============================\
=====
platform darwin -- Python 2.7.5 -- pytest-2.3.4 -- /usr/bin/python
collected 3 items

test_realistic_two_funcs.py:20: test_2_that_does_not
setup_module()
resources_a_setup()
test_2_that_does_not()
PASSED
teardown_module()
resources_a_teardown()

============================ 1 passed in 0.01 seconds =========================\
=====
```

This is a waste.
If the fixtures are quite lengthy, it can seriously slow down the test run unnecessarily.

Creating classes to separate fixture needs

You can relatively cleanly deal with this by either isolating tests requiring a resource to their own module, or to their own class.

I'll demonstrate the class solution to this problem.

Move the fixtures from module to class level, and move the tests that use the resource into the class.

```
from __future__ import print_function

def resource_a_setup():
    print('resources_a_setup()')

def resource_a_teardown():
    print('resources_a_teardown()')

class TestClass:

    @classmethod
    def setup_class(cls):
        print ('\nsetup_class()')
```

```
14          resource_a_setup()
15
16      @classmethod
17      def teardown_class(cls):
18          print ('\nteardown_class()')
19          resource_a_teardown()
20
21      def test_1_that_needs_resource_a(self):
22          print('\ntest_1_that_needs_resource_a()')
23
24  def test_2_that_does_not():
25      print('\ntest_2_that_does_not()')
```

Now we can run the test in isolation without expensive and unnecessary resource setup/cleanup.

```
1  $ py.test -s -v test_separation_of_fixtures.py::test_2_that_does_not
2  ================================ test session starts ============================\
3  =====
4  platform darwin -- Python 2.7.5 -- pytest-2.3.4 -- /usr/bin/python
5  collected 2 items
6
7  test_separation_of_fixtures.py:25: test_2_that_does_not
8  test_2_that_does_not()
9  PASSED
10
11 ============================== 1 passed in 0.01 seconds =========================\
12 =====
```

And the resource is still dealt with correctly when we do need it.

```
1  $ py.test -s -v test_separation_of_fixtures.py
2  ================================ test session starts ============================\
3  =====
4  platform darwin -- Python 2.7.5 -- pytest-2.3.4 -- /usr/bin/python
5  collected 2 items
6
7  test_separation_of_fixtures.py:21: TestClass.test_1_that_needs_resource_a
8  setup_class()
9  resources_a_setup()
10
11 test_1_that_needs_resource_a()
12 PASSED
```

```
13 teardown_class()
14 resources_a_teardown()
15
16 test_separation_of_fixtures.py:25: test_2_that_does_not
17 test_2_that_does_not()
18 PASSED
19
20 ============================ 2 passed in 0.01 seconds ========================\
21 =====
```

For lots and lots of circumstances, resource handling in this manner is completely sufficient.

However, I do believe that the pytest fixture mechanism (which I'll cover in my next post), is a more elegant and scalable solution to the problem.

pytest fixtures easy example

In pytest xUnit style fixtures[34], I presented a problem where:

- Two tests exist in a test file.
- One uses a resource.
- The other doesn't.
- Module level fixtures don't work if you just want to run the one function that doesn't use the resource.

I then presented class level fixtures[35] as a way to solve the separation problem.

In this post, I'll use pytest fixtures to solve the same problem.
I'm not going into details of all the goodies you get with pytest fixtures.
I'll just stick to solving this problem.
In future posts, I'll go into more details about pytest fixtures.

the problem

Here's the code that caused us trouble last time:

```
1  from __future__ import print_function
2
3  def resource_a_setup():
4      print('resources_a_setup()')
5
6  def resource_a_teardown():
7      print('resources_a_teardown()')
8
9  def setup_module(module):
10     print('\nsetup_module()')
11     resource_a_setup()
12
13 def teardown_module(module):
14     print('\nteardown_module()')
```

[34] http://pythontesting.net/framework/pytest/pytest-xunit-style-fixtures/
[35] http://pythontesting.net/framework/pytest/pytest-xunit-style-fixtures/#creating-classes-to-separate-fixture-needs

```
15      resource_a_teardown()
16
17  def test_1_that_needs_resource_a():
18      print('test_1_that_needs_resource_a()')
19
20  def test_2_that_does_not():
21      print('\ntest_2_that_does_not()')
```

The problem is that if I want to just run 'test_2_that_does_not()', the fixture functions for resource_a are called, even though I don't need them to be called.

```
1   $ py.test -s -v test_realistic_two_funcs.py::test_2_that_does_not
2   ================= test session starts ==================
3   platform darwin -- Python 2.7.5 -- pytest-2.3.4 -- /usr/bin/python
4   collected 3 items
5
6   test_realistic_two_funcs.py:20: test_2_that_does_not
7   setup_module()
8   resources_a_setup()
9
10  test_2_that_does_not()
11  PASSED
12  teardown_module()
13  resources_a_teardown()
14
15  ================ 1 passed in 0.01 seconds ===============
```

the pytest fixture solution

Instead of moving the resource_a related fixtures and tests into a class, we:

1. Import pytest
2. Use the pytest fixture decorator to specify 'resource_a_setup()' as a fixture.
3. Specify the fixture as module scope, so if two tests need it, it will still only have setup/teardown called once.
4. Specify 'resource_a_teardown()' as a finalizer for 'resource_a_setup()'. To do this, we need to add a 'request' param to the setup function. Also, note that the finalizer function can be very local to the setup function, even defined within it.
5. Include 'resource_a_setup' in the param list for tests that use resource_a.

I'll also add one more test function that uses the resource, to prove that module scope works.

For pytest fixtures to work, steps #1, #2 and #5 are all that are really needed.
Step #3 is only needed if you want to modify the default (which is 'function').
Step #4 is only needed if you want to include a teardown function.

So, here's my code.

```
from __future__ import print_function
import pytest

@pytest.fixture(scope='module')
def resource_a_setup(request):
    print('\nresources_a_setup()')
    def resource_a_teardown():
        print('\nresources_a_teardown()')
    request.addfinalizer(resource_a_teardown)

def test_1_that_needs_resource_a(resource_a_setup):
    print('test_1_that_needs_resource_a()')

def test_2_that_does_not():
    print('\ntest_2_that_does_not()')

def test_3_that_does(resource_a_setup):
    print('\ntest_3_that_does()')
```

Running only 'test_2_that_does_not':

```
$ py.test -s -v test_three_funcs.py::test_2_that_does_not
================= test session starts ==================
platform darwin -- Python 2.7.5 -- pytest-2.3.4 -- /usr/bin/python
collected 4 items

test_three_funcs.py:14: test_2_that_does_not
test_2_that_does_not()
PASSED

=============== 1 passed in 0.01 seconds ===============
```

Running everything:

```
$ py.test -s -v test_three_funcs.py
================= test session starts ==================
platform darwin -- Python 2.7.5 -- pytest-2.3.4 -- /usr/bin/python
collected 3 items

test_three_funcs.py:11: test_1_that_needs_resource_a
resources_a_setup()
test_1_that_needs_resource_a()
PASSED
test_three_funcs.py:14: test_2_that_does_not
test_2_that_does_not()
PASSED
test_three_funcs.py:17: test_3_that_does
test_3_that_does()
PASSED
resources_a_teardown()


================ 3 passed in 0.01 seconds ===============
```

some benefits of pytest fixtures

Right away we can see some cool benefits.

- It's obvious which tests are using a resource, as the resource is listed in the test param list.
- I don't have to artificially create classes (or move tests from one file to another) just to separate fixture usage.
- The teardown code is tightly coupled with the setup code for one resource.
- Scope for the lifetime of the resource is specified at the location of the resource setup code. This ends up being a huge benefit when you want to fiddle with scope to save time on testing. If everything starts going haywire, it's a one line change to specify function scope, and have setup/teardown run around every function/method.
- It's less code. The pytest solution is smaller than the class solution.

an even smaller example

I stated earlier in the solution, that steps #3 and #4 are optional.
Let's take a look at the simplified code if we just go with the defaults.

```
from __future__ import print_function
import pytest

@pytest.fixture()
def resource_a():
    print('\nresources_a() "setup"')

def test_1_that_needs_resource_a(resource_a):
    print('test_1_that_needs_resource_a()')

def test_2_that_does_not():
    print('\ntest_2_that_does_not()')

def test_3_that_does(resource_a):
    print('test_3_that_does()')
```

The difference?

- No teardown code (finalizer). So no need for a request param for the setup func.
- No scope specified. The default will call 'resource_a' before every func/method that needs it.
- Oh yeah. I also shortened the resource name. Dropping off the '_setup'.

This shortened version is more typical of how I would start writing my test code.

I only add finalizers (teardown) if necessary for the resource.
It is cool to note that only the resource fixture has to care about the finalizer.
You can add it if you need to, and the change needed is only to the setup fixture code.

I also usually am ok with function level scoping at first.
I pay attention to run times and realistic needed scoping for resources, and fiddle with scope if necessary.
And again, this fiddling is isolated to the resource fixture code.
The tests don't have to change to support different scoping.

So, here's my test run:

```
$ py.test -s -v test_three_funcs_small.py
================= test session starts ==================
platform darwin -- Python 2.7.5 -- pytest-2.3.4 -- /usr/bin/python
collected 3 items

test_three_funcs_small.py:8: test_1_that_needs_resource_a
resources_a() "setup"
test_1_that_needs_resource_a()
PASSED
test_three_funcs_small.py:11: test_2_that_does_not
test_2_that_does_not()
PASSED
test_three_funcs_small.py:14: test_3_that_does
resources_a() "setup"
test_3_that_does()
PASSED

================ 3 passed in 0.03 seconds ===============
```

wrap up

My main goal for this post was to show that using pytest fixtures is **at least** as easy as using the class fixture solution to separate fixture usage.

I hope I've demonstrated that.

pytest fixtures nuts and bolts

I'd like to wrap up this recent series of pytest fixture posts by presenting my version of some sort of reference.
Since this post is running a bit long, here are some links to the content buried in here.

- Bare bones example
- Three ways to use a fixture
  - Name it, usefixtures, and autouse.
  - usefixtures example
- Fixture features
  - Return value
  - Finalizer is teardown
  - Request object
  - Scope
  - Params
    - Toy example
    - Real example
  - Autouse
  - Multiple fixtures
  - Modularity: fixtures using other fixtures
- Experimental and still to cover
  - yield_fixture
  - ids

Since I'm planning on using this for my own reference, I'll throw a couple more links in here at the top to stuff I find useful regarding pytest fixtures.

- pytest tutorial[36], my original introduction post. I still use it as a reference.
- pytest support for xunit style fixtures[37], setup/teardown for function, method, module, class. I still use this old style sometimes.
- pytest API[38], specifically, the fixtures and requests part

And… while I'm throwing links around, here are the other posts in the series:

[36] http://pythontesting.net/framework/pytest/pytest-introduction/
[37] http://pythontesting.net/framework/pytest/pytest-xunit-style-fixtures/
[38] http://pytest.org/latest/builtin.html#fixtures-and-requests

- part 1: pytest fixtures[39]
- part 2: xunit style pytest fixtures[40]
- part 3: pytest fixtures easy example[41]

I'm trying to keep the examples in this post kind of small-ish.
However, I do run the risk of being too terse. (or too verbose).
It's also possible that the order I've laid things out is odd. I've done a lot of copy/paste from code editor and bash window to get this post put together.

Please let me know:

- if I've mucked up some copy/paste and there is something bizarre in here.
- if I've been too terse or unclear.
- if what I'm stating is completely wrong. **Especially this one**
- if you actually made it to the end without wanting to throw something at me.

Note about commmon code

For all the examples, the test file I'm running has this at the top:

```
1 from __future__ import print_function
2 import pytest
```

However, I'm not going to copy it into every code block below.
I'm also running each example with: ‘

```
1 $py.test -s file_name.py
```

Bare bones example

Here's a super basic fixture, and a couple tests that use it.

[39] http://pythontesting.net/framework/pytest/pytest-fixtures/
[40] http://pythontesting.net/framework/pytest/pytest-xunit-style-fixtures/
[41] http://pythontesting.net/framework/pytest/pytest-fixtures-easy-example/

```
@pytest.fixture()
def before():
    print('\nbefore each test')

def test_1(before):
    print('test_1()')

def test_2(before):
    print('test_2()')
```

With the default parameters for 'pytest.fixture()', the fixture is going to be called for every test that names the fixture in it's parameter list.

Output:

```
before each test
test_1()
.
before each test
test_2()
.
```

Three ways to use a fixture

1. **name it** from the test.
   Just like the top example
2. **usefixtures decorator**
   You can mark a test or a test class with 'pytest.mark.usefixtures()' and include a list of fixtures to be used with the test or class of tests.
   This is especially convenient when dealing with test classes.
   It also is useful when converting unittest classes to use pytest fixtures.
   I'll give an example shortly.
3. **autouse**
   Powerful, but possibly dangerous.
   Covered in the next section.

Usefixtures example

Here's a quick example of the decorator.
The first example can be written like this: '

```
@pytest.mark.usefixtures("before")
def test_1():
    print('test_1()')

@pytest.mark.usefixtures("before")
def test_2():
    print('test_2()')
```

Or, this:

```
class Test:
    @pytest.mark.usefixtures("before")
    def test_1(self):
        print('test_1()')

    @pytest.mark.usefixtures("before")
    def test_2(self):
        print('test_2()')
```

Or, this:

```
@pytest.mark.usefixtures("before")
class Test:
    def test_1(self):
        print('test_1()')

    def test_2(self):
        print('test_2()')
```

All with the same effect.

Fixture features

If I fill in the default parameters for ‘pytest.fixture()’ and add a request param to my fixture, it looks like this, but doesn’t run any different. ‘

```
@pytest.fixture(scope='function', params=None, autouse=False)
def before(request):
    print('\nbefore()')
    return None
```

Now lets take a look at these features.

- Return value
- Finalizer is teardown
- Request objects
- Scope
- Params
  - Toy example
  - Real example
- Autouse
- Multiple fixtures
- Modularity: fixtures using other fixtures

Return value

In the bare bones example, the fixture returns 'None'.
That's becuase it's just some code I want to run before my test, like traditional setup functions from nose[42] or unittest[43].

However, you can return anything you want from the fixture function.
If your fixture is setting up some data, or reading a file, or opening a connection to a database, then access to that data or resources is what you ought to return from the fixture.

Returning some data from a fixture.

```
@pytest.fixture()
def some_data():
    data = {'foo':1, 'bar':2, 'baz':3}
    return data

def test_foo(some_data):
    assert some_data['foo'] == 1
```

Returning a database object:

[42] http://pythontesting.net/framework/nose/nose-introduction
[43] http://pythontesting.net/framework/unittest/unittest-introduction

```
@pytest.fixture()
def cheese_db(request):
    print('\n[setup] cheese_db, connect to db')
    # code to connect to your db
    a_dictionary_for_now = {'Brie': 'No.', 'Camenbert': 'Ah! We have Camenbert, \
yessir.'}
    def fin():
        print('\n[teardown] cheese_db finalizer, disconnect from db')
    request.addfinalizer(fin)
    return a_dictionary_for_now

def test_cheese_database(cheese_db):
    print('in test_cheese_database()')
    for variety in cheese_db.keys():
        print('%s : %s' % (variety, cheese_db[variety]))

def test_brie(cheese_db):
    print('in test_brie()')
    assert cheese_db['Brie'] == 'No.'

def test_camenbert(cheese_db):
    print('in test_camenbert()')
    assert cheese_db['Camenbert'] != 'No.'
```

Finalizer is teardown

In the previous code example, the 'cheese_db' fixture has this bit of code:

```
def cheese_db(request):
    ...
    def fin():
        print('\n[teardown] cheese_db finalizer, disconnect from db')
    request.addfinalizer(fin)
    ...
```

The 'fin' function is acting as the 'teardown' for the fixture.
There's nothing special about the name.
You can name it 'teardown' or 'cheese_db_teardown' or 'something_else'.
It doesn't matter.

The finalizer is called after all of the tests that use the fixture.
If you've used parameterized fixtures, the finalizer is called between instances of the parameterized fixture changes.

Scope

Scope controls how often a fixture gets called. The default is “function”.
Here are the options for scope:

function Run once per test

class Run once per class of tests

module Run once per module

session Run once per session Since the default scope is “function”, the cheese db example will open and close the db for every test.

```
[setup] cheese_db, connect to db
in test_cheese_database()
Camenbert : Ah! We have Camenbert, yessir.
Brie : No.
.
[teardown] cheese_db finalizer, disconnect from db

[setup] cheese_db, connect to db
in test_brie()
.
[teardown] cheese_db finalizer, disconnect from db

[setup] cheese_db, connect to db
in test_camenbert()
.
[teardown] cheese_db finalizer, disconnect from db
```

This doesn’t really make sense. Especially if connecting is a time consuming operation.

Change the scope:

```
@pytest.fixture(scope="module")
def cheese_db(request):
  ...
```

And let’s re-run it:

```
[setup] cheese_db, connect to db
in test_cheese_database()
Camenbert : Ah! We have Camenbert, yessir.
Brie : No.
.in test_brie()
.in test_camenbert()
.
[teardown] cheese_db finalizer, disconnect from db
```

That's better.

Request objects

In the cheese db example, the fixture includes a request parameter. You need the request paramter to a fixture to add a finilizer.

However, it has other uses too.

In the example below, I'm showing the use (well, printing stuff) of some of the items. See pytest API[44] for a full list.

```
@pytest.fixture()
def my_fixture(request):
    print('\n-----------------')
    print('fixturename : %s' % request.fixturename)
    print('scope       : %s' % request.scope)
    print('function    : %s' % request.function.__name__)
    print('cls         : %s' % request.cls)
    print('module      : %s' % request.module.__name__)
    print('fspath      : %s' % request.fspath)
    print('-----------------')

    if request.function.__name__ == 'test_three':
        request.applymarker(pytest.mark.xfail)

def test_one(my_fixture):
    print('test_one():')

class TestClass():
    def test_two(self, my_fixture):
        print('test_two()')

```

[44] http://pytest.org/latest/builtin.html#fixtures-and-requests

```
22 def test_three(my_fixture):
23     print('test_three()')
24     assert False
```

Params

An optional parameter to the fixture decorator is ‘params’.
It defaults to ‘None’.

For each value in params, the fixture will be called with request.param filled in with that value.
Tests that use the fixture will be called once FOR EACH value in params.

Toy example

An example is in order here.

This first example is a silly one, but does show the mechanics, and the utility of both the -v flag and how well py.test deals with failures of paramterized tests.

```
1 @pytest.fixture( params=[1,2,3] )
2 def test_data(request):
3     return request.param
4
5 def test_not_2(test_data):
6     print('test_data: %s' % test_data)
7     assert test_data != 2
```

This should run ‘test_not_2’ three times, and fail when 2 is passed in.
I’ll run it both without and with the -v flag.

```
1  > py.test test_params.py
2  ============================= test session starts ==============================
3  platform win32 -- Python 2.7.2 -- pytest-2.4.2
4  collected 3 items
5
6  test_params.py .F.
7
8  =================================== FAILURES ===================================
9  ____________________________________ test_not_2[2] ____________________________________
10
11 test_data = 2
12
13     def test_not_2(test_data):
```

```
14          print('test_data: %s' % test_data)
15  >       assert test_data != 2
16  E       assert 2 != 2
17
18  test_params.py:10: AssertionError
19  ------------------------------ Captured stdout ------------------------------
20  test_data: 2
21  ==================== 1 failed, 2 passed in 0.02 seconds ====================
22  > py.test -v test_params.py
23  ============================= test session starts ==============================
24  platform win32 -- Python 2.7.2 -- pytest-2.4.2 -- C:\Python27\python2.7.exe
25  collecting ... collected 3 items
26
27  test_params.py:8: test_not_2[1] PASSED
28  test_params.py:8: test_not_2[2] FAILED
29  test_params.py:8: test_not_2[3] PASSED
30
31  ================================== FAILURES ===================================
32  _____________________________ test_not_2[2] ______________________________
33
34  test_data = 2
35
36      def test_not_2(test_data):
37          print('test_data: %s' % test_data)
38  >       assert test_data != 2
39  E       assert 2 != 2
40
41  test_params.py:10: AssertionError
42  ------------------------------ Captured stdout ------------------------------
43  test_data: 2
44  ==================== 1 failed, 2 passed in 0.02 seconds ====================
```

Real example

Now for a more real world usage of paramterization, input and expected output.

Here's a rewrite of the markdown test from the pytest introduction[45] post.
The 'run_markdown' function is a software API adapter[46], which takes care of calling the markdown script on the command line.

[45] http://pythontesting.net/framework/pytest/pytest-introduction/#testing_markdown
[46] http://pythontesting.net/strategy/software-api-cli-interface-adapters

```
from markdown_adapter import run_markdown

@pytest.fixture( params=[
    # tuple with (input, expectedOutput)
    ('regular text'   , 'regular text</p>'),
    ('*em tags*'      , '<p><em>em tags</em></p>'),
    ('**strong tags**', '<p><strong>strong tags</strong></p>')
    ])
def test_data(request):
    return request.param

def test_markdown(test_data):
    (the_input, the_expected_output) = test_data
    the_output = run_markdown(the_input)
    print('\ntest_markdown():')
    print('  input   : %s' % the_input)
    print('  output  : %s' % the_output)
    print('  expected: %s' % the_expected_output)
    assert the_output == the_expected_output
```

The output makes it clear that the one test 'test_markdown' is called 3 times.
Of course, the print statements are unnecessary.
I left them in for demo purposes.

```
> py.test -s  test_markdown.py
============================= test session starts ================
platform win32 -- Python 2.7.2 -- pytest-2.4.2
collected 3 items

test_markdown.py
test_markdown():
  input   : regular text
  output  : regular text
  expected: regular text
.
test_markdown():
  input   : *em tags*
  output  : <em>em tags</em>
  expected: <em>em tags</em>
.
test_markdown():
  input   : **strong tags**
```

```
  output   : <strong>strong tags</strong>
  expected: <strong>strong tags</strong>
.

========================== 3 passed in 0.11 seconds ==============
```

Normally, you don't need the print statements to see what's going on.

I'm going to take the print statements out, and change the 'expected' string of the last input value to show how py.test is quite helpful in pointing out the data set that fails.

```
@pytest.fixture( params=[
    # tuple with (input, expectedOutput)
    ('regular text'   , 'regular text</p>'),
    ('*em tags*'      , '<p><em>em tags</em></p>'),
    ('**strong tags**', '<p><strong>strong tags</em></p>')
    ])
def test_data(request):
    return request.param

def test_markdown(test_data):
    (the_input, the_expected_output) = test_data
    the_output = run_markdown(the_input)
    assert the_output == the_expected_output
```

output:

```
> py.test test_markdown2.py
============================= test session starts ============================
platform win32 -- Python 2.7.2 -- pytest-2.4.2
collected 3 items

test_markdown2.py ..F

================================= FAILURES =================================
_____________________________ test_markdown[test_data2] _____________________________

test_data = ('**strong tags**', '<strong>strong tags</em></p>')

    def test_markdown(test_data):
        (the_input, the_expected_output) = test_data
        the_output = run_markdown(the_input)
```

```
>       assert the_output == the_expected_output
E       assert '<p><strong>s...</strong></p>' == '<p><strong>st...tags</em></p>'
E         - <p><strong>strong tags</strong>
E         ?                          ^^^^^^
E         + <strong>strong tags</em>
E         ?                          ^^

test_markdown2.py:17: AssertionError
====================== 1 failed, 2 passed in 0.11 seconds ======================
```

Autouse

An optional parameter to the fixture decorator is 'autouse'.
It defaults to 'False'.

With the value of 'False', tests that wish to use the fixture need to either name it in their parameter list, or have a use fixtures decorator applied to the test.

See the first two items in section 'three ways to use a fixture' for more information.

With the value set to 'True', all tests in this session just use the fixture automatically.

Yes, with great power comes great responsibility[47].

So use it carefully.

However, it is quite handy in places where you would have used the xunit style setup_module[48]

For our example, let's just say I've got some reporting code I'd like to run at the top of module and at the top of a test funcion.

The tests themselves don't need a handle to the fixtures, since they aren't returning any data.

```
@pytest.fixture(scope="module", autouse=True)
def mod_header(request):
    print('\n-----------------')
    print('user         : %s' % getpass.getuser())
    print('module       : %s' % request.module.__name__)
    print('-----------------')

@pytest.fixture(scope="function", autouse=True)
def func_header(request):
    print('\n-----------------')
    print('function     : %s' % request.function.__name__)
    print('time         : %s' % time.asctime())
```

[47] http://en.wikiquote.org/wiki/Stan_Lee

[48] http://pythontesting.net/framework/pytest/pytest-xunit-style-fixtures/

```
13      print('-----------------')
14
15  def test_one():
16      print('in test_one()')
17
18  def test_two():
19      print('in test_two()')
```

output:

```
1   > py.test -s test_autouse.py
2   ============================= test session starts ==============================
3   platform win32 -- Python 2.7.2 -- pytest-2.4.2
4   collected 2 items
5
6   test_autouse.py
7   -----------------
8   user         : okken
9   module       : test_autouse
10  -----------------
11
12  -----------------
13  function     : test_one
14  time         : Tue Feb 04 17:31:20 2014
15  -----------------
16  in test_one()
17  .
18  -----------------
19  function     : test_two
20  time         : Tue Feb 04 17:31:20 2014
21  -----------------
22  in test_two()
23  .
24
25  ========================== 2 passed in 0.02 seconds ===========================
```

Multiple fixtures

In the examples I've used so far, tests only are using at most one named fixture. You can use more.

Simple example:

```
@pytest.fixture(scope="module")
def foo(request):
    print('\nfoo setup - module fixture')
    def fin():
        print('foo teardown - module fixture')
    request.addfinalizer(fin)

@pytest.fixture()
def bar(request):
    print('bar setup - function fixture')
    def fin():
        print('bar teardown - function fixture')
    request.addfinalizer(fin)

@pytest.fixture()
def baz(request):
    print('baz setup - function fixture')
    def fin():
        print('baz teardown - function fixture')
    request.addfinalizer(fin)

def test_one(foo, bar, baz):
    print('in test_one()')

def test_two(foo, bar, baz):
    print('in test_two()')
```

output:

```
> py.test -s test_multiple.py
============================= test session starts ==============================
platform win32 -- Python 2.7.2 -- pytest-2.4.2
collected 2 items

test_multiple.py
foo setup - module fixture
bar setup - function fixture
baz setup - function fixture
in test_one()
.baz teardown - function fixture
bar teardown - function fixture
bar setup - function fixture
```

```
baz setup - function fixture
in test_two()
.baz teardown - function fixture
bar teardown - function fixture
foo teardown - module fixture

========================== 2 passed in 0.01 seconds ==========================
```

Modularity: fixtures using other fixtures

Tests can use one or more fixture.

Fixtures themselves can also use one or more fixtures.

I'll rewrite the previous example, but instead of having the tests include all foo, bar, and baz fixtures, I'll chain them together.

And one more wrinkle, 'test_two' will only include 'bar'.

```
@pytest.fixture(scope="module")
def foo(request):
    print('\nfoo setup - module fixture')
    def fin():
        print('foo teardown - module fixture')
    request.addfinalizer(fin)

@pytest.fixture()
def bar(request, foo):
    print('bar setup - function fixture')
    def fin():
        print('bar teardown - function fixture')
    request.addfinalizer(fin)

@pytest.fixture()
def baz(request, bar):
    print('baz setup - function fixture')
    def fin():
        print('baz teardown - function fixture')
    request.addfinalizer(fin)

def test_one(baz):
    print('in test_one()')

def test_two(bar): # only use bar
    print('in test_two()')
```

output:

```
> py.test -s test_modular.py
============================= test session starts ==============================
platform win32 -- Python 2.7.2 -- pytest-2.4.2
collected 2 items

test_modular.py
foo setup - module fixture
bar setup - function fixture
baz setup - function fixture
in test_one()
.baz teardown - function fixture
bar teardown - function fixture
bar setup - function fixture
in test_two()
.bar teardown - function fixture
foo teardown - module fixture

========================== 2 passed in 0.02 seconds ===========================
```

Experimental and still to cover

In this section, I'm listing the experimental features, and features I haven't fully tested and/or don't quite understand yet.

- yield_fixture
- ids

yield_fixture

Thank you Johannes for pointed out this feature in the comments.

You can use 'yield_fixture' instead of the 'fixture' decorator for functions that yield their value rather than returning them.

The benefits are:

- It works like a context manager.
- You don't have to register a teardown function via addfinalizer.
- Therfore, you don't have to include the request parameter just for the addfinalizer function.

Caveats:

- It's still "experimental", so supposedly the syntax/behavior might change in the future.
- Probably more, but I haven't used it much to know what else to be careful of.

Here's a quick example. (Yep. Staight from the comment.)

```
@pytest.yield_fixture(scope="module")
def cheese_db():
    print('\n[setup] cheese_db, connect to db')
    a_dictionary_for_now = {'Brie': 'No.', 'Camenbert': 'Ah! We have Camenbert, \
yessir.'}
    yield a_dictionary_for_now
    print('\n[teardown] cheese_db finalizer, disconnect from db')

def test_brie(cheese_db):
    print('in test_brie()')
    assert cheese_db['Brie'] == 'No.'

def test_camenbert(cheese_db):
    print('in test_camenbert()')
    assert cheese_db['Camenbert'] != 'No.'
```

output:

```
$ py.test -s pytestfixtures/test_yield.py
====================== test session starts ======================
platform darwin -- Python 2.7.5 -- py-1.4.20 -- pytest-2.5.2
collected 2 items

pytestfixtures/test_yield.py
[setup] cheese_db, connect to db
in test_brie()
.in test_camenbert()
.
[teardown] cheese_db finalizer, disconnect from db

=================== 2 passed in 0.01 seconds ====================
```

WARNING: My recommendation is to be aware of this feature, but use 'addfinalizer' for production test code.

This is a cool feature. But since it's still listed as 'experimental', and I haven't done much testing with it or testing of it, I can't in good conscience recommend it's use.

Hey pytest devs: Let me know if this WARNING is too strong.

I DO recommend you use it IFF you are either solo or on a small team where you are able to easily change the test code in the future if the syntax/behavior changes in future pytest releases.

ids

More information at pytest.org[49]

I played around with this a bit, but couldn't get anything to work.
I'm sure I was just doing something wrong.
If anyone has a working example they could share, please do.

[49] http://pytest.org/latest/builtin.html#fixtures-and-requests

pytest session scoped fixtures

In pytest fixtures nuts and bolts[50], I noted that you can specify session scope so that a fixture will only run once per test session and be available across multiple test functions, classes, and modules.

In this post, I'm going to show a simple example so you can see it in action.

- A separate file for fixtures, conftest.py
- Simple example of session scope fixtures
- Mixing function, module, and session scope
- Taking it further
- What are you using session fixtures for?

Here's the table from the previous post[51]:

function Run once per test

class Run once per class of tests

module Run once per module

session Run once per session

A separate file for fixtures, conftest.py

With function, class, and module scope, it is completely reasonable for the fixture code to be in the same file as the tests.
But now suddenly, with session, that doesn't make sense anymore.

We can put them in conftest.py. This is a special named file that pytest looks for.
The documentation[52] says that it's for local plugins, but we can use it for local fixtures as well. See the pytest.org site[53] for placement and scope of conftest.py.

[50] http://pythontesting.net/framework/pytest/pytest-fixtures-nuts-bolts
[51] http://pythontesting.net/framework/pytest/pytest-fixtures-nuts-bolts/#scope
[52] http://pytest.org/latest/plugins.html#conftest-py-plugins
[53] http://pytest.org/latest/plugins.html#conftest-py-plugins

Simple example of session scope fixtures

I think it's clearest to just see this in action.

I've got 4 files:

- conftest.py
  - 2 fixtures
  - my_own_session_run_at_beginning, an autouse fixture with session scope
  - some_resource, a normal non-autouse fixture with session scope
- test_alpha.py
  - 2 simple test functions
  - test_alpha_1, has no named fixtures
  - test_alpha_2, has one named fixture, some_resource
- test_beta.py
  - similar to test_alpha.py, but with unittest based tests
- test_gamma.py
  - similar to test_alpha.py, but with class based tests

conftest.py:

```
import pytest

@pytest.fixture(scope="session", autouse=True)
def my_own_session_run_at_beginning(request):
    print('\nIn my_own_session_run_at_beginning()')

    def my_own_session_run_at_end():
            print('In my_own_session_run_at_end()')
    request.addfinalizer(my_own_session_run_at_end)


@pytest.fixture(scope="session")
def some_resource(request):
    print('\nIn some_resource()')

    def some_resource_fin():
            print('\nIn some_resource_fin()')
    request.addfinalizer(some_resource_fin)
```

test_alpha.py:

```
def test_alpha_1():
    print('\nIn test_alpha_1()')

def test_alpha_2(some_resource):
    print('\nIn test_alpha_2()')
```

test_beta.py:

```
import unittest
import pytest

class BetaTest(unittest.TestCase):
    def test_unit_beta_1(self):
        print('\nIn test_unit_beta_1()')

    @pytest.mark.usefixtures('some_resource')
    def test_unit_beta_2(self):
        print('\nIn test_unit_beta_2()')
```

test_gamma.py:

```
class TestGamma:
    def test_gamma_1(self):
        print('\nIn test_gamma_1()')

    def test_gamma_2(self, some_resource):
        print('\nIn test_gamma_2()')
```

Output

Run with **`pytest -s -v`**

```
============================= test session starts ==============================
platform darwin -- Python 2.7.5 -- py-1.4.20 -- pytest-2.5.2 -- /usr/bin/python
collecting ... collected 6 items

test_alpha.py:1: test_alpha_1
In my_own_session_run_at_beginning()

In test_alpha_1()
PASSED
```

```
test_alpha.py:4: test_alpha_2
In some_resource()

In test_alpha_2()
PASSED
test_beta.py:5: BetaTest.test_unit_beta_1
In test_unit_beta_1()
PASSED
test_beta.py:8: BetaTest.test_unit_beta_2
In test_unit_beta_2()
PASSED
test_gamma.py:2: TestGamma.test_gamma_1
In test_gamma_1()
PASSED
test_gamma.py:5: TestGamma.test_gamma_2
In test_gamma_2()
PASSED
In some_resource_fin()
In my_own_session_run_at_end()
========================== 6 passed in 0.04 seconds ==========================
```

Mixing function, module, and session scope

Let's say I've got:

- a function scope fixture 'resource_c'
- that uses a module scoped fixture 'fixture_b'
- that uses a session scoped fixture 'fixture_a'

This all works fine.
Also in this example, I've added a few autouse fixtures just for fun.

conftest.py:

```
import pytest

@pytest.fixture(scope="session")
def resource_a(request):
    print('In resource_a()')

    def resource_a_fin():
            print('\nIn resource_a_fin()')
    request.addfinalizer(resource_a_fin)

@pytest.fixture(scope="module")
def resource_b(request, resource_a):
    print('In resource_b()')

    def resource_b_fin():
            print('\nIn resource_b_fin()')
    request.addfinalizer(resource_b_fin)

@pytest.fixture(scope="function")
def resource_c(request, resource_b):
    print('In resource_c()')

    def resource_c_fin():
            print('\nIn resource_c_fin()')
    request.addfinalizer(resource_c_fin)

# these are just some fun dividiers to make the output pretty
# completely unnecessary, I was just playing with autouse fixtures
@pytest.fixture(scope="function", autouse=True)
def divider_function(request):
    print('\n        --- function %s() start ---' % request.function.__name__)
    def fin():
            print('        --- function %s() done ---' % request.function.__name\
__)
    request.addfinalizer(fin)

@pytest.fixture(scope="module", autouse=True)
def divider_module(request):
    print('\n    ------- module %s start ---------' % request.module.__name__)
    def fin():
            print('    ------- module %s done ---------' % request.module.__name\
__)
```

```
43      request.addfinalizer(fin)
44
45  @pytest.fixture(scope="session", autouse=True)
46  def divider_session(request):
47      print('\n----------- session start ---------------')
48      def fin():
49              print('----------- session done ---------------')
50      request.addfinalizer(fin)
```

test_one_two.py:

```
1   def test_one(resource_c):
2       print('In test_one()')
3
4   def test_two(resource_c):
5       print('\nIn test_two()')
```

test_three_four.py:

```
1   def test_three(resource_c):
2       print('\nIn test_three()')
3
4   def test_four(resource_c):
5       print('\nIn test_four()')
```

This seems reasonable to me.
What do you think will happen?

output:

```
1   $ py.test -s -v
2   ==================================== test session starts ======================\
3   =============
4   platform darwin -- Python 2.7.5 -- py-1.4.20 -- pytest-2.5.2 -- /usr/bin/python
5   collected 4 items
6
7   test_one_two.py:1: test_one
8   ----------- session start ---------------
9
10      ------- module test_one_two start ---------
11
12          --- function test_one() start ---
```

```
13 In resource_a()
14 In resource_b()
15 In resource_c()
16 In test_one()
17 PASSED
18 In resource_c_fin()
19         --- function test_one() done ---
20
21 test_one_two.py:4: test_two
22         --- function test_two() start ---
23 In resource_c()
24
25 In test_two()
26 PASSED
27 In resource_c_fin()
28         --- function test_two() done ---
29
30 In resource_b_fin()
31     ------- module test_one_two done ---------
32
33 test_three_four.py:1: test_three
34     ------- module test_three_four start ---------
35
36         --- function test_three() start ---
37 In resource_b()
38 In resource_c()
39
40 In test_three()
41 PASSED
42 In resource_c_fin()
43         --- function test_three() done ---
44
45 test_three_four.py:4: test_four
46         --- function test_four() start ---
47 In resource_c()
48
49 In test_four()
50 PASSED
51 In resource_c_fin()
52         --- function test_four() done ---
53
54 In resource_b_fin()
```

```
55      ------- module test_three_four done ---------
56
57  In resource_a_fin()
58  ----------- session done --------------
59
60
61  ================================ 4 passed in 0.02 seconds ====================\
62  =============
```

WARNING: you gotta use bigger and bigger scope

If you do this in the wrong order, things go haywire.

Let's swap scope on a couple of items.

conftest.py:

```
1   ...
2   @pytest.fixture(scope="module")   # session -> module
3   def resource_a(request):
4       print('In resource_a()')
5
6       def resource_a_fin():
7               print('\nIn resource_a_fin()')
8       request.addfinalizer(resource_a_fin)
9
10  @pytest.fixture(scope="session")   # module -> session
11  def resource_b(request, resource_a):
12  ...
```

We will get some warning like this (or several):

```
1   E           ScopeMismatchError: You tried to access the 'module' scoped fixture \
2   'resource_a'
3   ... with a 'session' scoped request object, involved factories
4   E           conftest.py:18:  def resource_c(request, resource_b)
5   E           conftest.py:10:  def resource_b(request, resource_a)
6   E           conftest.py:2:  def resource_a(request)
```

So. Don't do that.

Warning applies to built in fixtures

Pytest includes some built in fixtures[54]. I believe all of them are function scoped.
This means that you cannot use them from anything other than functions or function scoped fixtures.

Taking it further

The code I've shown is for simple run at the beginning and end type fixtures.
However, there's more you can do with session fixtures.
The pytest.org site has a cool example, A session-fixture which can look at all collected tests[55].

What are you using session fixtures for?

I'd love to hear examples and use cases for session fixtures.
Please leave a comment or let me know @brianokken[56] of how you are using them.

[54] http://pytest.org/latest/builtin.html#builtin-fixtures-function-arguments
[55] http://pytest.org/latest/example/special.html
[56] https://twitter.com/brianokken