Python unit testing and mocking

Write tests first, then code.

Python unit testing

Prior to circa Python 2.6/7, module `unittest2` held the modern functionality needed. Module `unittest` is all that's needed.

Python `unittest` is "me-too" in the world of xUnit. (This is a very big world; everybody does it: Java, C#, even C/C++.) So unit test customs and practices are well known and understood.

Python unit testing

It appears to exist for Python 2.6 on our appliances

```
Python 2.6.6 (r266:84292, Jan 22 2014, 09:42:36)
[GCC 4.4.7 20120313 (Red Hat 4.4.7-4)] on linux 32
>>> import unittest
>>> dir( unittest )
['FunctionTestCase', 'TestCase', 'TestLoader', 'TestProgram',
'TestResult', 'TestSuite', 'TextTestRunner', '_CmpToKey',
'_TextTestResult', '_WriteInDecorator', '__all__', '__author__',
'__builtins__', '__doc__', '__email__', '__file__', '__metaclass__',
'__name__', '__package__', '__unittest__', '__version__', '_makeLoader',
'_strclass', 'defaultTestLoader', 'findTestCase, 'getTestCaseNames',
'main', 'makeSuite', 'os', 'sys', 'time', 'traceback', 'types']
```

Python unit testing -- nose

Nose is a superset of `unittest` and exists for Python 2.6 just as `unittest`.

Mostly, nose improves test discovery and that's what PerfectSearch appears to use it for.

So `unittest` should be acceptable as it's not incompatible or even different, it's not "either-or."

Python unit testing

There are:

- module set-up and tear-down (`@classmethod`)
- test case set-up and tear-down
- test cases

unittest basics

```
import unittest

class XyzTest( unittest.TestCase ):      # (inherits Python unit test framework)

     def testThing( self ):              # (one test case)
          x = 9
          self.assertTrue( x == 9 )

if __name__ == '__main__':               # (so this can be run from the command line)
     unittest.main()
```

`unittest` set-up and tear-down

```
class XyzTest( unittest.TestCase ):

    def setUp( self ):
        self.havingFun = True

    def tearDown( self ):
        self.havingFun = False

    def testHavingFun( self ):
        self.assertTrue( self.havingFun )
```

Works no differently from an ordinary Python class (`self`, etc.).

`unittest` set-up and tear down (class-level)

```
class XyzTest( unittest.TestCase ):
     @classmethod
     def setUpClass( XyzTest ):
          # create a test file on /tmp, etc.
          ( XyzTest.fd, XyzTest.path ) = tempfile.mkstemp()
          os.write( XyzTest.fd, 'something useful' )
          os.close( XyzTest.fd )
     @classmethod
     def tearDownClass( XyzTest ):
          # clean test file from /tmp, etc.
          os.remove( XyzTest.path )
```

No `self` possible here! If you need a “global” variable, use class name.

unittest One last, useful goodie...

```
class XyzTest( unittest.TestCase ):

  @unittest.skip( 'This is broken for now.' )
  def testMethod( self ):
    pass
```

Test-driven development (TDD)

Short and crude:

> TDD is you writing a tiny bit of test that fails, fixing that failure, then writing another test that fails.

TDD in Python is little different from any other language.

It requires mastery of unit test idiom.

Mocking is extremely useful in focusing on real problem.

Hamcrest BDD* idiom available in Python, but not necessary.

(*) Behavior-driven development

Python mocking

Most of Python 3 mocking is supported in Python 2.6 and 2.7

Mocking is a simple concept that appears complicated on the outside.

Subject under test (SUT), the class or module being tested, usually, its instantiated object.

Unit-testing/mocking terminology

Fakes are objects with working, but a very minimal or zero-grade implementation. (Okay, that's easy.)

Stubs provide canned answers to calls made by the SUT usually not responding in any way to details outside their intended purpose. (Hmmm…, sure.)

Mocks are objects with pre-programmed expectations forming a specification of calls they are expected to receive from the SUT. (Aargh!)

Fakes and stubs replace functionality the SUT consumes, mocks do not.

(Well, not exactly, wait for it…)

Fake example (1), auth_conditions

```
class InRole:
  def __init__( self, role ):
    self.roles = role
    pass
  def check( self ):
    return True
  def __call__( self, aList ):
    pass
```

```
class nullAuthorize(object):
  def __init__( self, arg1=None, arg2=None ):
    self.arg1 = arg1
    self.arg2 = arg2
  def __call__( self, function ):
    def innerFunction( *args, **kwargs ):
      return function( *args, **kwargs )
    return innerFunction
  def authorize( valid, handler=None ):
    def _authorize( func, self, *args, **kwargs ):
      return func( self, *args, **kwargs )
    return nullAuthorize( _authorize )
```

```
@authorize( InRole( [ 'system', 'search' ] ) )
def putSet( self, sessionid, definitionname ):
  payload = getRequestBody()
  ...
```

Fake example (2), `pylons.request`

```
class usage( object ):
  def __init__( self ):
    pass
  def write( self, message, append=False ):
    if append:
      print 'LOG: ' + message

body = ""
params = {}
environ = {}
environ[ 'psUsageLogEntry' ] = usage()
```

Stub example, sessions

```
class SessionsController( object ):
  def __init__( self ):
    pass

  def showallsessions( self, sessionid=None ):
    return '<sessions><session id="group_searchappliance@default" state="dormant" users...'

  def showsession( self, sessionid=None ):
    return '<sessions><session id ...<sessionobj name="testSet" state="dormant" users="1">...'
```

Python mocking (okay, back to it!)

The Python world is less terminologically precise, everything's called a mock; *patch* is also used.

Aargh? Think of *fake* and *stub*. These replace actual production objects. However, mocking is when you don't write a special version of a production class, but use a framework to wrap the real one and intercept particular uses.

Typically, a rule of thumb is never to mock something you do not own. But, in Python mocking this is not held so much.

Mock example (1)

```
import unittest
import os
from jack import Jack

class JackTest( unittest.TestCase )
  def setUp( self ):
    self.saveOsRemove = os.remove
    os.remove = myRemove
  def tearDown( self ):
    os.remove = self.saveOsRemove
  def testCase( self ):
    jack = Jack()
    jack.removeMe( '/path/filename' )

def myRemove( filepath ):
  print 'Didn\'t actually try to remove', filepath
```

```
import os

class Jack( object ):
  def removeMe( self, path ):
    os.remove( path )
```

(Brutal mocking here...

...not in fact using any framework.)

Mock example (2)

```
>>> from mock import Mock
>>> class Bar( object ):
...   def something( self ):
...         return 'something'
...
>>> bar = Bar()
>>> bar.something()
'something'
>>> bar.something = Mock()
>>> bar.something.return_value = 'foo'
>>> bar.something()
'foo'
```

Mock example (3)

```
>>> class Foo( object ):
...   def something( self ):
...         return 'something'
...
>>> foo = Foo()
>>> foo.something()
'something'
>>> foo.something = Mock()
>>> foo.something.side_effect = [ 'foo', 'bar', 'baz' ]
>>> foo.something()
'foo'
>>> foo.something()
'bar'
>>> foo.something()
'baz'
```

Mock example (4)

```
class Foo( object ):
  def something( self ):
      return 'something'


def mySideEffect( *args, **kwargs ):
  if args[ 0 ] == 42:
      return 'foo'
  elif args[ 0 ] == 43:
      return 'bar'
  elif kwargs[ 'foo' ] == 7:
      return 'baz is shazaz'
```

```
>>> import foo
>>> foo = Foo()
>>> foo.something = Mock()
>>> foo.something.side_effect = mySideEffect
>>> foo.something( 42 )
'foo'
>>> foo.something( 43 )
'bar'
>>> foo.something( -1, foo=7 )
'baz is shazaz'
```

Mock example (5)

```
class SetsTest( unittest.TestCase ):
  @patch( 'searchui.controllers.sessions.SessionsController.showallsessions' )
  def testGetWorld( self, showallsessionPatchedReturn ):
    showallsessionPatchedReturn.return_value = loadFullShowAllSessionsReturn()*
    sets.getRequestBody = Mock()
    sets.getRequestBody.side_effect = [ COMPOUND1_DEF, COMPOUND2_DEF ]

    sessionid = 'session1'
    dfn = '__query1__'
    self.setsController.putSet( sessionid, dfn )
    assert( pylons.response.status_int == 200 )
    dfn = '__query2__'
    self.setsController.putSet( sessionid, dfn )
    assert( pylons.response.status_int == 200 )

    actual = self.setsController.getSets()
```

(*) eschewing `showallsessions()` stub in favor of special, local one

Python mocking problems

You must ensure you're patching the object used by the SUT. This is not always obvious. To make it more likely successful,

a. In production code (SUT), avoid using `from … import as`.
b. When accessing an object, use the full path to it.
c. In test code, perform import identically.

It's sometimes easier to play small tricks. These influence production code, but can also be a good thing. For example, your SUT consumes something complex in a dependent module. Create a function that does this for the class; mock that function from your test code. It's easier, less brittle and a bit self-documenting.

Python mocking problems

Mocking is inherently brittle.

This means that refactoring your code will necessarily entail a revamp of your test code because it will be broken.

Often, however, your test code is broken in ways that you want to know about anyway (unless it’s a total rewrite).

In short, when your test code blows up, it’s frequently a sign your implementation has trouble as long as you tested behavior in the first place.