Python for Test Automation

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME
1.0	August 2012		SF

Contents

Chapter 1

Basic Tools

1.1 Lesson objectives

Despite Python's "batteries included" philosophy we frequently want to install 3rd party packages for use in our Python programs. Python comes with a facility for automatically installing modules but all packages are installed globally and on some systems it requires adminstrative privileges. We can use standard tools to create isolated Python environments and install 3rd party packages into them.

1.1.1 virtualenv

Virtualenv is a tool that has swiftly become a community standard in the Python world - so much so that official support for virtualenv will ship with Python 3.3 when it is released. Consider the typical ways in which you might install a Python package or application

Django installation

```
$ tar xzvf Django-1.3.tar.gz
$ cd Django-1.3
$ sudo python setup.py install
```

The disadvantage of this approach is that Django ends up installed globally. The built-in installer uses setuptools and places the django directory containing importable code in your shared system packages directory. On Windows, for example, this might be in `C:\Python2.7\Lib\site-packages`.

Using a single global location to install Python modules means that it is impossible to to have more than one version of your package installed. It also means that you have to have administrative permissions to install Python modules. Fortunately there's a better way to manage installation of Python modules.

virtualenv

```
`virtualenv` is a tool to create isolated Python environments.
```

Ian Bicking's tool virtualenv allows us to create Python environments that are isolated from one another. It supports all three major OS platforms (Windows/Mac/Linux) and has become a defacto standard with integration in other tools such as mod_wsgi and the pip installer.

To install virtualenv we can use easy_install, the installer provided by setuptools. This should be automatically installed with Python on Mac and Linux but Windows users may have to download and run the setuptools installer for your version of Python - you can find it at http://pypi.python.org/pypi/setuptools

Installing virtualenv

```
$ easy_install virtualenv
```

`virtualenv` works by creating a Python environment, copying or symnlinking your Python executable and creating a local directory in which to install Python modules. This is best demonstrated by example:

Let's make a virtualenv

```
$ virtualenv PROJ1
New python executable in PROJ1/bin/python
Installing setuptools............done.
$ which python              # The virtual env is not yet activated
/usr/bin/python
$ source PROJ1/bin/activate # Activate using the source command
(PROJ1)$ which python       # Notice the prompt indicates the active env
/home/simeon/PROJ1/bin/python
```

The `virtualenv` command is used to create a virtualenvironment in the directory specified. Virtualenvironments are just directories with a Python environment - they can be activated by running the activate script using the `source` command on the activate script on Mac/Linux or by running the `activate.bat` on windows.

The prompt is modified to show the currently active virtualenvironment and now running commands like `python` or `easy_install` runs a local copy in the `bin` directory of the virtualenvironment instead of the global shared executable.

Install Nose

```
(PROJ1)$ easy_install nose
Searching for nose
Reading http://pypi.python.org/simple/nose/
Reading http://somethingaboutorange.com/mrl/projects/nose/
Reading http://readthedocs.org/docs/nose/
Best match: nose 1.1.2
Downloading http://pypi.python.org/packages/source/n/nose/nose-1.1.2.tar.gz#md5=144 ↩
    f237b615e23f21f6a50b2183aa817
Processing nose-1.1.2.tar.gz
... snip ...
Processing dependencies for nose
Finished processing dependencies for nose
```

Notice that easy_install was not run with administrative permissions. Not only is the Python code put in the right place (the site-packages directory of the currently activated virtualenv) but utilities like the *nosetests* test runner are put in a `bin` folder in the currently activated environment as well.

1.2 PRACTICE ACTIVITY

1. Make sure your environment is all set up. Go ahead and install setuptools if on Windows - other OS's should already have setuptools installed. You can check by running the `easy_install` command in your shell - if you've got it you already have setuptools installed.
2. Install virtualenv using easy_install.
3. Create a virtualenv called `nose`. Install `nose` in it using easy_install. Go ahead and easy_install the "coverage" module as we'll be needing that too.
4. You can check to make sure nose is installed correctly by opening a Python console and running `import nose`.

Tip

Mac users who have recently upgraded to Lion seem to have a variety of problems with `virtualenv` and `easy_install`. Most of these problems seem to be resolved by running all updates for xcode.

1.2.1 virtualenvwrapper

Doug Hellman's `virtualenvwrapper` is a set of helper commands for managing virtualenvs. It is written in bash so unfortunately is not available for Windows - although Windows users might check out https://github.com/davidmarble/virtualenvwrapper-win for an implementation in batch scripts.

The idea behind virtualenvwrapper is to store all virtualenvs in a specific place (by default `~/.virtualenv` but configurable) and add utility commands for creating, activating, manually adding locations, and listing the contents of a particular virtualenv. We won't be covering installation and usage of virtualenvwrapper in this course as it doesn't modify the functionality of the underlying `virtualenv` tool, but many Python hackers find virtualenvwrapper an indispensable part of their toolkit to manage their development environment.

1.2.2 PIP

You might also be interested in replacing `easy_install` with `pip`. Pip is another Ian Bicking tool that has become an indispensable part of many Python devopers toolkit. Because using pip at a basic level is identical to `easy_install` we aren't requiring it as part of the class and won't cover installation - I'll just encourage you to

```
$ sudo easy_install pip
```

and use pip instead of easy_install. Pip includes features like default .egg unzipping, uninstallation, listing installed packages, native support for virtual environments, installing directly from source control, building a new environment from a requirements file and much more. See http://www.pip-installer.org/en/latest/index.html for more details.

1.3 *Basic Tools* Follow-up

This lesson introduced tools that should become part of your basic workflow as a Python developer. Particularly with a focus on testing, the ability to create isolated Python environments, try out different versions of packages, and easily recreate a given environment is important. Now that we've got them, we'll use them to explore the ecosystem of 3rd party Python packages and developer tools.

Chapter 2

Nose

2.1 Lesson objectives

In this lesson we'll learn to use the 3rd party project `nose` to easily collect and run our unittests and doctests. The boilerplate that `unittest` requires to build suites and run them, including doctests and all the tests in various files is usually rendered unnecessary by nose. We'll explore a few of the options `nose` offers to make our testing life easier.

2.2 Organizing for nose

We're going to be working on a slightly larger code-base so we'd like to be organized. Nose doesn't impose much in the way of structure on the layout of our project and what structure it does impose is totally customizable. We'll just follow three rules:

1. Our main project directory will be a python package - it will have a *init*.py in it.
2. Our stand-alone tests will live in files called test_*.py subdirectories called "tests". Our subdirectories must be in a package or subpackage but won't have `__init__.py` files themselves and therefore won't be packages
3. All our import statements in our testing code will use absolute imports based on our top level package name.

2.3 Using Nose

If you followed the instructions in the last OOP lab you have the following directory layout:

Directory Listing

```
$ ls -l reports
total 8
-rw-r--r-- 1 simeon simeon    0 2012-08-19 18:46 __init__.py
-rw-r--r-- 1 simeon simeon    0 2012-08-19 18:46 report.py
drwxr-xr-x 2 simeon simeon 4096 2012-08-19 18:52 tests
```

The tests directory contains our test files - maybe only `test_report.py` at this point. We can go ahead and and delete any `unittest.main()` calls in our test files - we'll be using nose to run our tests. To run our tests we can use the test-runner script that ships with nose. Be sure to have your virtualenv that contains nose activated!

Running Tests

```
(nose)$ nosetests -x
E
======================================================================
ERROR: Failure: ImportError (No module named report)
```

Traceback (most recent call last): ... snip ... import report ImportError: No module named report

```
Ran 1 test in 0.002s

FAILED (errors=1)
```

The -x option to nosetests (or --stop) tells it to fail fast - as soon as a test fails or errors nose stops running tests. The error in this case is that previously we wrote our import statements assuming we were in the same directory as the code we were testing.

As our codebase grows this can become unwieldy - with many test scripts intermingled with application python files. We don't want anybody to accidentally import and run test code either - so nose makes sure the root level of our package is importable and suggests we use absolute imports starting with our package name. We'll need to fix our import statements like:

```
from reports import report
```

2.4 Nose options

Nose comes with a variety of options that customise it's behavior. It also has a cool plugin system and comes standard with a variety of useful plugins. Check out the man page or the official documentation at http://nose.readthedocs.org

You might investigate

- Verbosity with `-v`. If you wonder if your test is being run `-v` will list the test as well as its result.
- Add doctests with the `--with-doctest` flag. This will look for and run doctests in your application code. If you also have unittests written with doctest and stored in text files in the "tests" directory you might want to also use the `--doctest-tests` option.
- nose can generate extremely useful coverage reports. `--with-coverage` will be enough to get you started. As your codebase grows in size you'll probably want to generate the more readable .html reports - see the documentation for details

2.5 And much more

Nose also allows you to write tests that are simple functions using assert statements, provides additional testing methods, and provides finer granularity for test setUp (often referred to as test fixtures) than `unittest` provides. Nose also has facilities for tagging tests and running only tests with particular attributes and other niceties like the ability to open a debugger on the spot of failed tests or use the test runner to also generate profiling reports. We'll take advantage of a few of these facilities later on but I'd encourage you to explore them on your own as you complete the labs.

2.5.1 Lab

Use the options we've discussed to make sure all your tests run. Make sure you have a doctest embedded in a function or class and that it is captured. Generate a coverage report and check to see if you have 100% code coverage. Can you get to 100%?

2.6 *Nose* Follow-up

Nose should hopefully make finding, running, and perhaps even writing your tests considerably easier. There are other test frameworks that provide similar features - if you're already a fan of py.test you don't need to switch. If you've just been writing straight unittest code, however, using nose will hopefully reduce the friction involved in writing tests and encourage you to write more of them!

Chapter 3

Mocking and Monkeypatching

3.1 Lesson objectives

In this lesson we'll learn to construct mock objects with the `mock` package. Where we've structured our code to accept external depencies we can pass in mock objects. Where we haven't structured our code for easy testing we'll learn to "monkeypatch" - temporarily overwriting other objects in order to control their behavior. Python's dynamic nature makes monkeypatching easy, but that doesn't make it a good or safe practice so we'll learn some techniques for limiting the effects of our monkeypatches.

3.2 Monkeypatching

While we try to write code that has no dependencies on the external environment, sometimes we can't avoid it. If we can pass into our code all external dependencies in the forms of objects we can still write tests by providing mock objects that supply the expected behavior from the tested code's point of view but whose behavior is controllable.

If we have code that directly constructs external dependencies we're stuck "monkeypatching". The Python community has adopted the somewhat derogatory term monkeypatching to discourage the dubious practice of taking advantage of the dynamic nature of Python to overwrite the functionality of existing objects (including builtins) in order to inject the correct behavior.

3.2.1 3 Simple rules for monkeypatching

1. Don't!
2. But when you do, make your changes as simple and limited in scope as possible.
3. Immediately put things back the the way you found them.

The following examples demonstrate these rules. Consider first target code that reads and processes data from a file. If our design doesn't allow dependency injection we might have no choice but to monkeypatch the `open` builtin as demonstrated by the first function.

If we have an OOP design, on the other hand, we might be able to completely replace the method that use external resources instead of patching the builtin objects. Sometimes this is simpler, but it frequently means just giving up on testing the patched code - note that by replacing the `load` method below we are essentially giving up on testing it.

Monkey Patching Target

```
def count_customers(filename):
    # Cheesy version with list comprehension over file
    customers = [line.split(',')[0] for line in open(filename, "r")]
    return len(set(customers))
```

```
class CustomerDataFile(object):
    """ Typical usage:

    >>> cdf = CustomerDataFile("some-data.txt")
    >>> cdf.load()
    >>> cdf.count_customers()
    """
    def __init__(self, filename):
        self.filename = filename

    def load(self):
        """The data loading procedure might be very complicated (db,
        caching, lots of builtins, etc)."""
        self.data = [line.split(',').strip() for line in open(self.filename, "r")]
        print "In load"

    def count_customers(self):
            customers = [row[0] for row in self.data]
            return len(set(customers))
```

The tests we write then supply stubs for `open` builtin and for the `load` method of the sample code. In the second case we don't even bother writing a useful stub and instead modify the state of the class. Obviously with a more real-world example of sufficient complexity we are treading on dangerous ground by supplying an alternate implementation of `load` in our test!

Monkey Patching Test

```
import unittest
import StringIO
import monkeypatching1 as mp1

class MonkeyPatchingBuiltin(unittest.TestCase):
    def setUp(self):
        self.file = StringIO.StringIO("simeon,2012-08-24,4,1\nbob,2012-08-24,10,0\nsimeon ↩
            ,2012-08-25,15,21")
        # Save original version of open builtin
        self.open = __builtins__.open
        # Replace open with a fake
        __builtins__.open = lambda filename, mode="r": self.file

    def tearDown(self):
        # put back original version of open
        __builtins__.open = self.open

    def test_count_customers(self):
        self.assertEqual(mp1.count_customers("fakefilename.txt"), 2)

data = [['simeon','2012-08-24',4,1],
         ['bob','2012-08-24',10,0],
         ['simeon','2012-08-25',15,21]]

class MonkeyPatchingBoundMethod(unittest.TestCase):
    def test_patch_unbound_method(self):
        # Monkeypatching a function directly on a class is easy - try a bound method on an  ↩
            object!
        original_load = mp1.CustomerDataFile.load
        mp1.CustomerDataFile.load = lambda self: None
        cdf = mp1.CustomerDataFile("some-data.txt")
        # Load won't do anything, so we manually supply data
        cdf.load()
        cdf.data = data
        count = cdf.count_customers()
        self.assertEqual(count, 2)
```

```
        mp1.CustomerDataFile.load = original_load


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

3.3 Mocking (and a little more help with monkeypatching)

So far our monkeypatching just supplies stub objects or functions that provide the expected functionality - or enough of it to fool the code under test. Returning a StringIO object from the open method is an example of this. Technically `mocking` in testing circles refers to returning objects that can also verify that particular behavior occured. We could write more complicated objects like this ourselves, but frequently Pythonistas use mocking and patching frameworks if they have to do significant amounts of mock object work.

I like the `mock` library. You can install it into your *nose* virtualenv with easy_install or pip. Using mock we can create magical objects that are infinitely malleable and any actions taken on our mock objects will be logged.

```
>>> import mock
>>> m = mock.Mock()
>>> m.foo()
>>> m.foo.called
True
>>> m.complicated_method(x=1, y=2)
>>> m.complicated_method.call_args
call(y=2, x=1)
```

Obviously we can supply objects that accept arbitrary calls with arbitrary arguments and we can check after the fact to make sure that the appropriate methods were called with the appropriate arguments. Of course we might also want to build objects (and chains of objects) that have known return values in order to stand in for the object they are mocking. Mock objects let us directly assign any properties we want and assign to the `return_value` attribute of any mock methods we want.

```
>>> m2 = mock.Mock()
>>> m2.hour = 10 # m2 is an object with an "hour" attribute set to 10
>>> m.datetime.now.return_value = m2 # m.datetime.now() will return m2
>>> m.datetime.now().hour # looks like a candidate for mocking the datetime library
10
```

The `mock` library also comes with context managers and decorators that can be used to do monkey patching. Importantly using the context managers or decorators allows `mock` to take care of rule #3 and always undo patching automatically when the test code has finished running.

```
>>> import datetime
>>> def ampm():
...     hour = datetime.datetime.now().hour
...     if hour < 12:
...         return "AM"
...     else:
...         return "PM"
>>> with mock.patch("datetime.datetime") as m:
...     m2 = mock.Mock(hour=13)
...     m.now.return_value = m2
...     print ampm()
PM
>>> datetime.datetime.now().hour
7
```

3.4 Common Problems

3.4.1 Immutable implementations

The builtin base types are immutable as are types written in C in the stdlib. This does not mean that they produce values that are immutable, it means you can't alter the value of the base types themselves.

```
>>> dict.foo = 1
Traceback (most recent call last):
  ...
TypeError: can't set attributes of built-in/extension type 'dict'
```

One workaround is to replace the builtin types with a class that inherits from the builtin type - the class will have the same functionality as it's base type but can be extended.

3.4.2 Misunderstanding the import statement

Consider the following module:

foo.py

```
def f1():
    print "original f"
    return 1

def f2():
    print f1()
```

and the failed attempt to monkeypatch

test.py

```
from foo import f1, f2
f1 = lambda: 2
f2()
```

Do you understand why this is a failure? How could we fix `test.py` to correctly modify `f2` by monkeypatching `f1`?

3.5 Lab

We'll return to testing our `reports` lab. Do you have tests for your class that handles the zip uri? Try writing tests using mock to monkeypatch the zipfile module so you can test your code.

3.6 *Mocking and Monkeypatching* Follow-up

Writing code to support testing is a huge topic - we haven't really scratched the surface of the capabilities of the `mock` module. Be sure to develope your monkeypatching and mocking skills - but always keep in mind that good design makes monkeypatching unnecessary and mocking easy. Better a little more effort in design than hours spent debugging monkeypatches and tangled networks of fake objects!

Chapter 4

Context managers and more decorators

4.1 Lesson objectives

Using the `mock` library may have given you a feel for the usefulness of advanced features like context managers and decorators in reducing boilerplate and prevent errors. But context managers can be a little cumbersome (as we will see) and decorators are hard to write functionally. With a little help from classes and generators we'll simplify the task of writing decorators and contextmanagers.

4.2 Context managers

The default way of writing a context manager is to supply a class that implements an `__enter__` method and an `__exit__` method. If you want to pass arguments to your context manager you can define and `__init__` method as well. Your class will be created as a result of a `with` statement and the `__exit__` method is guaranteed to be run even if an exception is raised and in fact receives arguments indicating exceptions that occured in the managed code block.

This is definitely a useful feature since you can write cleanup code that otherwise must be (but might not) added to a `finally` block by the user of your class. Using a context manager allows the builtin file type to close itself, for instance.

The matter is more complicated if you want to produce an object that will be received by the optional `as` portion of the `with` statement. This isn't your contextmanager at all but is supposed to be a separate object. Properly writing a well behaved contextmanager than will require two classes and two magic method implementations. This seems like a lot of boilerplate and fortunately `contextlib` can help us write simple context managers simply.

Consider the following generator function:

```
>>> def looks_contexty():
...     print "before"
...     try:
...         yield "as object"
...     finally:
...         print "guaranteed after"
```

This function when called would generate an object who upon the first .next() would run the first part. When the second .next() call occurs it will run the code in the `finally` clause. This sounds a lot like a context manager and the `contextlib.contextmanager` decorator can in fact make it one.

```
>>> import contextlib
>>> @contextlib.contextmanager
... def cm():
...     print "before"
...     try:
...         yield "as object"
```

```
...     except:
...         print "an error occured"
...     finally:
...         print "no matter what"
...
>>> with cm() as foo:
...     print foo
...
before
as object
no matter what

>>> with cm():
...     print 1/0
before
an error occured
no matter what
```

4.2.1 Lab

I frequently find the `timeit` class annoying to use in the interactive console since it wants to be passed an evalable string of Python code. Can you write a timing context manager that works like:

```
>>> with mytimeit("It took %s milliseconds"):
...     pass
It took 1 milliseconds
```

4.3 Decorators, revisited

You may have noticed when we discussed closures that function closures sound remarkably like objects - they are produced by a factory that ties together state (the enclosed variables) and a function. You may further have noticed when we discussed emulation that objects can be callable, emulating functions. Putting those two facts together you can probably figure out how to write a class based version of a decorator yourself.

```
class Logger(object):
>>> class Logger(object):
...     def __init__(self, func):
...         self.func = func
...     def __call__(self, *args, **kwargs):
...         print args, kwargs
...         return self.func(*args, **kwargs)
...
>>> @Logger
... def foo(x, y):
...     return x - y
...
>>> foo(5, 1)
(5, 1) {}
4
```

An object of the class Logger is instanciated (that's what happens when you call a class), accepting the function it is decorating. The Logger object is now standing in for our function so when we call it we get the `__call__` method which can call the original function that we saved in our `__init__`.

Tip
You can write decorators that are configurable but doing so functionally takes two levels of closures, doing so with an object takes only one and the resulting code is much more Pythonic.

4.4 Lab

Rewrite the decorators for your `reports` lab using classes. Do your tests still pass?

Tip

As of Python 2.6 it's now legal to decorate classes as well as functions! Can you think of any use cases for class decorators?

4.5 Lab

Move all your data source classes to a new file called `sources.py`. Modify your `accept_sources` decorator to process URI's generically by looking up the protocol in a registry data structure that maps protocols to the appropriate class. Can you write a class based decorator that when applied to classes return them unmodified but adds their protocol mapping to the registry variable? If you can figure out how to write a configurable decorator (I like http://www.artima.com/weblogs/-viewpost.jsp?thread=240845 for a good description) registering a custom class for a custom uri (or multiple uri's) should look like:

```
from reports import register

@register("zip://")
@register("gz://")
class MyZipFile(object):
    pass
```

4.6 *Context managers and more decorators* Follow-up

Python's advanced facilities are fun, but sometimes a little mindblowing. Always use the easiest implementation you can get away with! Afer completing this lesson the idea of writing decorators may have gotten a bit easier and writing contextmanagers with contextlib and yield should seem downright trivial.