System Development with Python: Week 1

Christopher Barker

UW Continuing Education

March 26, 2013

Table of Contents

Class Structure

- Lecture
- Exercises
- Lab Time

Testing

I hope I don't need to tell you why testing is important

I'll focus on testing your Python code – not another system

Mostly unit testing

And an introduction to some of the tools

Python Testing Taxonomy

1. Unit Testing Tools
2. Mock Testing Tools
3. Fuzz Testing Tools
4. Web Testing Tools
5. Acceptance/Business Logic Testing Tools
6. GUI Testing Tools
7. Source Code Checking Tools
8. Code Coverage Tools
9. Continuous Integration Tools
10. Automatic Test Runners
11. Test Fixtures
12. Miscellaneous Python Testing Tools

`http://wiki.python.org/moin/PythonTestingToolsTaxonomy`

Testing

Regression Testing:

Regression testing is any type of software testing that seeks to uncover new software bugs, or regressions, in existing functional and non-functional areas of a system after changes

Unit Testing:

Unit testing is a method by which individual units of source code, sets of one or more computer program modules together with associated control data, usage procedures, and operating procedures, are tested to determine if they are fit for use

doctest

doctest

- In the stdlib
- Unique to Python?
- Literate programming
- Verify examples in in docs

`http://docs.python.org/library/doctest.html`

doctest example

```
def get_the_answer():
    """
    get_the_answer() -> return the answer to everything

    >>> print get_the_answer()
    42
    """
    return 42
```

(code/the_answer.py)

An exact dump of the command line output

http://docs.python.org/library/doctest.html

doctest uses

“ As mentioned in the introduction, doctest has grown to have three primary uses:

- Checking examples in docstrings.
- Regression testing.
- Executable documentation / literate testing.

These uses have different requirements, and it is important to distinguish them. In particular, filling your docstrings with obscure test cases makes for bad documentation.”

`http://docs.python.org/library/doctest.html`

running doctests

In `__name__ == "__main__"` block:

```
if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

(Tests the current module)

`http://docs.python.org/library/doctest.html`

running doctests

In an external file:

```
doctest.testfile("name_of_test_file.txt")
```

Tests the docs in the file (ReSt format...)

With a test runner: more on that later

running doctests

In a documentation system:

Sphinx:

Sphinx is a tool that makes it easy to create intelligent and beautiful documentation

– Lots of output options: html, pdf, epub, etc, etc...

– Can auto-generate docs from docstrings

– And run the doctests

`http://sphinx.pocoo.org/`

Code Checking

Not Really testing, but...

pychecker

`http://pychecker.sourceforge.net/`

pylint

`http://www.logilab.org/857/`

pyflakes

`http://pypi.python.org/pypi/pyflakes`

Will help you keep your source code clean

pep8

pep8.py

Tells you when you have code which doesn't follow PEP 8

```
pip install pep8
```

```
http://pypi.python.org/pypi/pep8/
```

LAB

doctests:

- doctest
  - Add doctests to the Circle class we've done in the last few classes.
  - Run it from the `if __name__ == "__main__"` clause (`code\circle.py`)
- Code style
  - Try running `pylint`, `pychecker` or `pyflakes` on it
  - Run pep8 on it
  - (or your own code)

Unit Testing

Gaining Traction

You need to test your code when you write it – why not preserve those tests?

And allow you to auto-run them later?

Test-Driven development:

- Write the tests before the code

Unit Testing

My thoughts:

Unit testing encourages clean, decoupled design

If it's hard to write unit tests for – it's not well designed

but...

"complete" test coverage is a fantasy

PyUnit

PyUnit: the stdlib unit testing framework

```
import unittest
```

More or less a port of Junit from Java

A bit verbose: you have to write classes & methods

unittest example

```
import random
import unittest

class TestSequenceFunctions(unittest.TestCase):

    def setUp(self):
        self.seq = range(10)

    def test_shuffle(self):
        # make sure the shuffled sequence does not lose any elem
        random.shuffle(self.seq)
        self.seq.sort()
        self.assertEqual(self.seq, range(10))

        # should raise an exception for an immutable sequence
        self.assertRaises(TypeError, random.shuffle, (1,2,3))
```

unittest example (cont)

```
    def test_choice(self):
        element = random.choice(self.seq)
        self.assertTrue(element in self.seq)

    def test_sample(self):
        with self.assertRaises(ValueError):
            random.sample(self.seq, 20)
        for element in random.sample(self.seq, 5):
            self.assertTrue(element in self.seq)

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

(code/unitest_example.py)

http://docs.python.org/library/unittest.html

unittest

Lots of good tutorials out there:

Google: "python unittest tutorial"

I learned from this one:

`http://www.diveintopython.net/unit_testing/index.html`

nose and pytest

Due to its Java heritage, unittest is kind of verbose

also no test discovery

(though unittest2 does add that...)

So folks invented nose and pytest

nose

nose

Is nicer testing for python

nose extends unittest to make testing easier.

```
$ pip install nose

$ nosetests unittest_example.py
```

http://nose.readthedocs.org/en/latest/

nose example

The same example – with nose

```
import random
import nose.tools

seq = range(10)

def test_shuffle():
    # make sure the shuffled sequence does not lose any elements
    random.shuffle(seq)
    seq.sort()
    assert seq == range(10)

@nose.tools.raises(TypeError)
def test_shuffle_immutable():
    # should raise an exception for an immutable sequence
    random.shuffle( (1,2,3) )
```

nose example (cont)

```
def test_choice():
    element = random.choice(seq)
    assert (element in seq)

def test_sample():
    for element in random.sample(seq, 5):
        assert element in seq

@nose.tools.raises(ValueError)
def test_sample_too_large():
    random.sample(seq, 20)
```

(code/test_random_nose.py)

pytest

pytest

- A mature full-featured testing tool
- Provides no-boilerplate testing
- Integrates many common testing methods

```
$ pip install pytest

$ py.test unittest_example.py
```

http://pytest.org/latest/

pytest example

The same example – with pytest

```
import random
import pytest

seq = range(10)

def test_shuffle():
    # make sure the shuffled sequence does not lose any elements
    random.shuffle(seq)
    seq.sort()
    assert seq == range(10)

def test_shuffle_immutable():
    pytest.raises(TypeError, random.shuffle,  (1,2,3) )
```

pytest example (cont)

```
def test_choice():
    element = random.choice(seq)
    assert (element in seq)

def test_sample():
    for element in random.sample(seq, 5):
        assert element in seq

def test_sample_too_large():
    with pytest.raises(ValueError):
        random.sample(seq, 20)
```

(code/test_random_pytest.py)

A Diversion:

Context Managers: the `with` statement

A class with `__enter__()` and `__exit__()` methods.

`__enter__()` is run before your block of code

`__exit__()` is run after your block of code

Can be used to setup/clenup before and after: open/closing files, db connections, etc

A Diversion

"PEP 343: the `with` statement"

– A.M. Kuchling

`http://docs.python.org/dev/whatsnew/2.6.html#pep-343-the-with-statement`

"Understanding Python's `with` statement"

– Fredrik Lundh

`http://effbot.org/zone/python-with-statement.htm`

"The Python `with` Statement by Example"

– Jeff Preshing

`http://preshing.com/20110920/the-python-with-statement-by-example`

Parameterized Tests

A whole set of inputs and outputs to test?
pytest has a nice way to do that (so does nose...)

```
import pytest
@pytest.mark.parametrize(("input", "expected"), [
    ("3+5", 8),
    ("2+4", 6),
    ("6*9", 42),
])
def test_eval(input, expected):
    assert eval(input) == expected
```

http://pytest.org/latest/example/parametrize.html

(code/test_pytest_parameter.py)

Test Coverage

coverage.py

Uses debugging hook to see which lines of code are actually executed – plugins exist for most (all?) test runners

```
pip install coverage
```

```
nosetests --with-coverage test_codingbat.py
```

http://nedbatchelder.com/code/coverage/

LAB

Unit Testing:

- unittest
  - Pick a `codingbat.com` example
  - Write a set of unit tests using `unittest` (`code\codingbat.py  codingbat_unittest.py`)
- pytest / nose
  - Test a `codingbat.com` with nose or pytest
  - Try doing test-driven development (`code\test_codingbat.py`)
- try running your circle doctest with nose, pytest (and/or add doctests to your codingbat example)
- try running `coverage` on your tests

Performance Testing

"Premature optimization is the root of all evil"

– Donald Knuth

Profiling/timing

You can't optimize your code without knowing where the bottlenecks are.

Smarter people than me have said they they are almost always wrong when they try to logically determine where the slow code is. (I know I am)

... and how to speed it up

time.clock()

The really easy way:

```
import time

start = time.clock()
  ... do_some_stuff ...
print "It took %f seconds to run"%(time.clock - start)
```

It works, it's easy, and it gives a gross approximation

(use `time.clock()`, rather than `time.time()`)

timeit

The good way:

```
import timeit

timeit.timeit( statement, setup=some_stuff)
```

It's kind of a pain, but gives meaningful results.
(can also be called on the command line)

`http://docs.python.org/library/timeit.html`

(`code/timing.py`)

%timeit

The easy and good way:

ipython:

```
In [52]: import timing

In [53]: %timeit timing.primes_stupid(5)
100000 loops, best of 3: 10.9 us per loop
```

Takes care of the setup/namespace stuff for you

http://ipython.org/ipython-doc/dev/interactive/tutorial.html

profiling

A profiler is a tool that describes the run time performance of a program, providing a variety of statistics

Helpful when you don't yet know where your bottlenecks are

The python profiler

```
python -m cProfile profile_example.py
```

spews some stats
(demo)

`http://docs.python.org/library/profile.html`

python profiler

What you get:

ncalls the number of calls.

tottime the total time spent in the given function (and excluding time made in calls to sub-functions),

percall the quotient of tottime divided by ncalls

cumtime the total time spent in this and all subfunctions (from invocation till exit). This figure is accurate even for recursive functions.

percall the quotient of cumtime divided by primitive calls

(demo: `python -m cProfile profile_example.py`)

python profiler

You can also dump to a file:

```
$ python -m cProfile -o profile_dump profile_example.py
```

This gives you a binary file you can examine with `pstats`:

demo: `$ python -m pstats`

pstats

Running pstats

```
$
$ python -m pstats
Welcome to the profile statistics browser.
% read profile_dump
profile_dump% stats
Wed Aug 29 16:21:39 2012    profile_dump

         51403 function calls in 0.032 seconds

   Random listing order was used

   ncalls  tottime  percall  cumtime  percall filename:lineno(fu
    51200    0.006    0.000    0.006    0.000 {method 'append' o
        1    0.000    0.000    0.032    0.032 profile_example.py
        1    0.001    0.001    0.032    0.032 profile_example.py
      100    0.022    0.000    0.027    0.000 profile_example.py
```

pstats commands

Commands:

help	help on pstats or particular command
stats	print the profile statistics
sort	sort by various data fields
strip	strips the leading path info from file names
callers	Print callers statistics
callees	Print callees statistics
quit	quits

Each has options to customize output

automating profile stats

`cProfile` and `pstats` are also modules

So you can script collection of profiles and stats

`http://docs.python.org/library/profile.html`

"Run Snake Run"

For a visual look at your profiling resluts:

`http://www.vrplumber.com/programming/runsnakerun/`

(pretty cool stuff!)

performance tips

Some common python performance issues:

`http://wiki.python.org/moin/PythonSpeed/PerformanceTips/`

(some nifty profiling tools described there, too)

LAB

Profiling lab

- run `timeit` on some code of yours (or timing.py, or..)
- run iPython's %timeit on the same code.
- try to make the factorial code in timing.py faster, and time the difference.
- write some code that tests one of the performance issues in: `http://wiki.python.org/moin/PythonSpeed/PerformanceTips` use one of the `timeits` to see if you can make a difference.
- try the profile tutorial at: `http://pysnippet.blogspot.com/2009/12/profiling-your-python-code.html`

Package Structure

You'll want to set up your project with good package structure

```
python setup.py develop
```

```
http://guide.python-distribute.org/
```

Wrap up

Hopefully you've got a bit of an idea how to do unit testing in Python.

And will now start doing it.

And have an idea how to profile and time your code.