System Development with Python: Week 1

Christopher Barker

UW Continuing Education

March 26, 2013

Table of Contents

Class Structure

- Lecture
- Labs
- Summary
- Project 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 the 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

Coding Style

Coding style really helps readability

Keep it consistent within your project

Options:

- Your company style guide
- PEP 8
  `http://www.python.org/dev/peps/pep-0008/`
- Another Established Style (Google's is pretty good)
  `http://google-styleguide.googlecode.com/svn/trunk/pyguide.html`

Coding Style

pep8.py

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

```
pip install pep8
```

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

LAB

doctests:

- doctest
  - Add doctests to the Circle class in the code dir
  - 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 first 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/cleanup 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/

Coding Bat

Coding Bat:

`http://codingbat.com/python`

Quicky little code excercises

Tells you what unit tests to write:

`http://codingbat.com/prob/p118406`

We'll use them for our lab

...and you might want to do a few for practice anyway...

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

Distributing

What if you need to distribute you own code?

Scripts

Libraries

Applications

Scripts

Often you can just copy, share, or check in the script to source control and call it good.

(But only if it's a single file, and doesn't need anything non-standard)

Scripts

When the script needs more than just the stdlib (or your company standard environment)

You have an application, not a script

Libraries

When you read the distutils docs, it's usually libraries they're talking about

Scripts + library is the same...

(`http://docs.python.org/distutils/`)

distutils

`distutils` makes it easy to do the easy stuff:

Distribute and install to multiple platforms, etc.

Even binaries, installers and compiled packages

(Except dependencies)

(`http://docs.python.org/distutils/`)

distutils basics

It's all in the `setup.py` file:

```
from distutils.core import setup
setup(name='Distutils',
      version='1.0',
      description='Python Distribution Utilities',
      author='Greg Ward',
      author_email='gward@python.net',
      url='http://www.python.org/sigs/distutils-sig/',
      packages=['distutils', 'distutils.command'],
     )
```

(http://docs.python.org/distutils/)

distutils basics

Once your setup.py is written, you can:

```
python setup.py ...

build          build everything needed to install
install        install everything from build directory
sdist          create a source distribution
               (tarball, zip file, etc.)
bdist          create a built (binary) distribution
bdist_rpm      create an RPM distribution
bdist_wininst create an executable installer for MS Windows
upload         upload binary package to PyPI
```

More Complex Packaging

For a complex package:

You want to use a well structured setup:

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

Package Structure

```
ProjectName/
    scripts/
    CHANGES.txt
    docs/
    LICENSE.txt
    setup.py
    project_package/
        __init__.py
      module1.py
      module2.py
    tests/
      test_module1.py
      test_module2.py
```

develop mode

While you are developing your package, Installing it is a pain.

But you want your code to be able to import, etc. as though it were installed.

`setup.py develop`

installs links to your code, rather than copies – so it looks like it's installed, but it's using the original source

You need `distribute` (or `setuptools`) to use it.

Applications

For a complete application:

- Web apps
- GUI apps

Multiple options:

- Virtualenv + RCS
- zc.buildout (`http://www.buildout.org/`)
- System packages (rpm, deb, ...)
- Bundles...

Bundles

Bundles are Python + all your code + plus all the dependencies – all in one single "bundle"

Most popular on Windows and OS-X

```
py2exe
py2app
pyinstaller
...
```

User doesn't even have to know it's python

Examples:

`http://www.bitpim.org/`

`http://response.restoration.noaa.gov/nucos`

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 an idea how to set up your package...

Next Week:

The python debugger pdb

– Jeff

And of course, your projects...

Homework

For the entire quarter, your homework is to work on your projects
For next week:

- Set up the package structure
- Put it in gitHub (or some RCS)
- Map out the design
- Develop a "plan of attack"
- Determine major third party packages you will need.
- Write at least a few unit tests (even if they all fail!)

Homework

A Plan of Attack

A plan of attack is an outline for how you intend to tackle you project – what you will do first, second, etc, and how long you think each component will take.

It doesn't need to be terribly detailed – perhaps mapped out week by week.

Note that at the end of the class you will present your work – and you should have *something* working – I'd rather see one working feature than 10 semi-completed but disfunctional ones.

Project Time!

- Do you have a project?
- Do you have a team?
- Do you know what modules you'll need?
- Let's get to work!