Testing Python

Understanding testing in the Python World

- Austin Godber
- Mail: godber@uberhip.com
- Twitter: @godber
- Source: http://github.com/godber/Python-Testing-Presentation

Motivation for Testing

- Does your code behave correctly to begin with?
- How about when you fix a bug?
- After you refactor?
- After everyone who wrote it is gone, how do you know what it was supposed to do in the first place?

Testing Topics

- Unit Testing
- Fixtures and Mocks
- Code Coverage and Code Quality
- Fuzz Testing
- Web Testing
- Acceptance/Functional Testing
- Regression Tests

Unit Testing - Description

- Testing the functionality of a small piece of application.
    - Does `MyInteger.add(2)` return what it is supposed to?
- Test Coverage
    - How much of the code is covered by existing tests?
        - Are all branches or code paths covered?
        - Are all functions/methods covered?

Unit Testing - Concepts

test fixture

setup and teardown for a collection of tests

test case

the code that actually implements the tests

test suite

a collection of test cases

test runner

interface for running test cases or suites

Unit Testing - Options

- Built In Solutions
  - `doctest`, `unittest`, `unittest2`
- 3rd Party Alternatives
  - `py.test`, `PyUnit`
- Testing Related Utils
  - `nose` - Helps run tests
  - `coverage`, `figleaf` - Testing Coverage
  - `pythonscope` - Test generation

Unit Testing - Example

An example using `unittest2`, `nose`, `coverage`, and `pythonscope`:

```
cd src/unittesting
nosetests --with-doctest -v
```

Running manually, without help from nose:

```
python -m doctest -v myinteger.py
PYTHONPATH=. python tests/test_myinteger.py
```

Unit Testing - Example

```
import myinteger, unittest2

class TestMyInteger(unittest2.TestCase):
    def test_add_simple(self):
        """Make sure single digit addition works"""
        i = myinteger.MyInteger(4)
        self.assertEqual(i.add(2), 6)

if __name__ == '__main__':
    unittest2.main()
```

Unit Testing - Assertions

The python unittest library is built around the set of assertions:

```
assertTrue, assertEqual, assertAlmostEqual, assertGreater, assertIn,
assert***Equal, assertRaises, assertIsNone, assertIs, assertIsInstance,
assertFalse
```

They take the arguments you would expect, plus a message.

```
assertTrue(i_return_true(), "I should return True")
assertEqual(two(), 2, "I should return two")
```

Remember Greg's Python Koans.

Example - Coverage

So how well does our existing test cover the MyInteger class?

```
PYTHONPATH=. coverage run --source=myinteger.py \
  tests/test_myinteger.py
coverage html
```

There is a nose plugin for coverage that probably simplifies this.

Example - Pythoscope

Lets generate the remaining tests:

```
pythoscope --init
pythoscope myinteger
```

Fixtures and Mocks

Fixtures

Help setup testing environment for a set of tests with similar requirements. Create test objects, load test data into db. A simple setup fixture is seen in the MyInteger example. The third party module, fixture, loads multiple database backends. There are also Django fixtures.

Mocks

Emulate actions or objects that are too expensive or disruptive to perform during every test run. e.g. `rocket.launch()` See the module Mock, mox

Example - Code Quality

pylint

Checks for errors, duplication, complexity and adherance to convention. Reports statistics on code composition and other metrics. Provides an overall rating. Very opinionated.

pep8

Does the code follow the PEP 8 style guide? Basic code style checking.

Fuzz Testing

Fuzz Testing

Throws garbage at your interfaces to see if it breaks.

Web Testing

- Standard Tools Apply: unittest, doctest
- Web Specific Tools
  - Browser Emulation
    - Django Test Client
  - Browser Drivers
    - Windmill, Selenium
  - Acceptance Test Frameworks, ATDD, BDD, Not strictly web related.
    - Lettuce, Pyccuracy

Django Testing - Basic

The built in Django test runner looks for doctests or unittests in:

- `models.py` - Runs docstrings and any subclass of unittest.TestCase.
- `tests.py` - Runs docstrings and any subclass of unittest.TestCase.

Rather than one huge `tests.py` file, you can make a `tests` directory, place your tests in that directory and then load the tests in `tests/__init__.py`. This is what I have done in the example application.

Django Testing - Run

How to run?:

```
cd src/djangoblog
python ./manage.py test blog
# or test all with
python ./manage.py test
```

Django Testing - Test Client

`django.test.client` emulates browser actions but can peek within the app itself.

- Does `get`, `post`, `put`, `delete`, `head`, `options` plus django auth `login` and `logout`.
- The response contains
  - Standard HTTP Stuff: `content`, `status_code` and dictionary of HTTP headers.
  - App Internal Stuff: `context`, `template`

Testing with Lettuce

This just a brief introduction as I know my understanding is severly lacking. `src/djangoblog/blog/features/` contains a basic example with the following issues.

- `world` is a global and I don't understand its scope
- Avoid repetitious and brittle code by using page_objects
- Browser drivers can be used in place of the django client, and thus can test non-django apps.

Run with:

```
python manage.py harvest
```

Continuous Integration

A CI system will build a project and run tests against it after every commit. This provides continuous feedback on the quality of your code.

- http://buildbot.net/trac
- http://greatbigcrane.com/
- http://hudson-ci.org/ (Java)