Survey of Unit-Testing Frameworks

by John Szakmeister and Tim Woods

Our Background

- Using Python for 7 years
- Unit-testing fanatics for 5 years

Agenda

- Why unit test?
- Talk about 3 frameworks:
  - unittest
  - nose
  - py.test

Why bother?

- Confidence!
- Automation
- To make failures more obvious
- To prevent regressions
- Aids debugging

Commonalities

- All the frameworks:
  - Follow the xUnit mentality
  - Verify via asserts

unittest

- In the standard library
- The de-facto standard unit test framework

Writing some tests...

- Need to subclass TestCase

```
import unittest

class TestFoo(unittest.TestCase):
    …
```

Write some tests...

- The simple way: test methods start with ‘test’
- Use assertions validate your results

Examples...

```
def testIsInt(self):
    self.assertTrue(foo.isInt(0))
    self.assertFalse(foo.isInt("not an int"))

def testMakeList(self):
    self.assertEquals([1,2,3],
                      foo.makeList(1, 2, 3))

def testDivide(self):
    self.assertEquals(0, foo.divide(0, 1))
    self.assertRaises(
        ZeroDivisionError, foo.divide, 1, 0)
```

Test Fixtures

- Useful if test cases need more “infrastructure”
- setUp() is called before each test method
- tearDown() is called after each test method

Example...

```
def setUp(self):
    # ... Any *common* set up required for your
    # test cases can go here
    self.db = create_db_connection()

def tearDown(self):
    # Clean up the fixture here
    self.db.close()
```

Running the Tests...

- Executing all tests within the file is easy
- Add this to the file:

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

- Collects all test cases, and executes them using the default console test runner

Example output

- On the command line, run:

```
:: PYTHONPATH=. python tests_unittest/test_foo.py
....
----------------------------------------------------------------------
Ran 4 tests in 0.000s

OK
```

Collecting All Tests...

- Need a suite to help collect the test cases
- In each file, do the following:

```python
def suite():
    testSuite = unittest.TestSuite()
    loader = unittest.TestLoader()

    for testCase in [TestFoo, TestFooBar]:
        testSuite.addTest(
            loader.loadTestsFromTestCase(testCase))
    return testSuite
```

(cont)

- Pull all the suites together in the __init__.py:

```
import unittest
import test_foo

def collectAll():
    allSuite = unittest.TestSuite(test_foo.suite())

    # If you have others, you can add them by
    # doing:
    #     allSuite.addTest(test_bar.suite())

    return allSuite
```

Finally, run all the tests

- Need to launch unittest.main(), but this time, tell it how to find the full test suite:

```
#!/usr/bin/env python
import unittest
import tests_unittest

unittest.main(
    defaultTest='tests_unittest.collectAll')
```

setuptools

- You can launch your tests via setuptools:

```
from setuptools import setup

setup(...,
      test_suite = 'tests_unittest.collectAll'
      )
```

Pros

- It’s in the standard library
- Writing the actual tests is easy

Cons

- Tedious to collect test suites, especially for a large code base
  - Michael Foord is working on this (check out the discover package)
- Not as easily extensible as other frameworks
  - Others have done it though. See the testtools project in Launchpad.

Future Improvements

- unittest is vastly improved in Python 2.7
  - Test discovery
  - Skipping
  - Expected Failures
  - assertRaises using 'with' statement
  - New assert methods, and much more

nose

- Written by Jason Pellerin

Install nose

- Available from http://somethingaboutorange.com/mrl/projects/nose/
- Use easy_install:

```
easy_install nose
```

Just write tests

- Finds test files, functions, classes, methods
  - test or Test on a word boundary
  - Customize with regular expression
- No need to write suites
- Use package structure to organize tests
  - Add __init__.py to directory

Write a test

- Create a file, test_foo.py:

```
from nose.tools import *

import foo

def test_isInt():
    assert_true(foo.isInt(0))
```

Use assertions

- Provided in ‘nose.tools’
- Same asserts as unittest, in PEP 8 style
- Also provides:
  - ok_ => assert
  - eq_ => assert_equals

Use assertions

```
def test_isInt():
    assert_true(foo.isInt(0))
    assert_false(foo.isInt("not an int"))

def test_makeList():
    assert_equals([1,2,3], foo.makeList(1, 2, 3))

def test_divide():
    eq_(0, foo.divide(0, 1))
```

Check exceptions

- Verify that test function raises exception
- Use raises decorator:

```
@raises(ZeroDivisionError)
def test_divide_by_zero():
    foo.divide(1, 0)
```

Test fixtures

- Use with_setup decorator
- Can be used for simple setup
- More complex cases should probably use a test class
- Fixtures per package, module, class

Test fixtures

```
_db = None

def setup_func():
    global _db
    _db = create_db_connection()

def teardown_func():
    global _db
    _db.close()

@with_setup(setup_func, teardown_func)
def test_with_fixture():
    ok_(isinstance(_db, DummyConnection))
```

Test classes

- Define class that matches test regexp
- Define test methods in the class
- Optionally define setup and teardown

Test classes

```
class TestFoo():
  def setup(self):
      # Fixture setup
      self.db = create_db_connection()

  def teardown(self):
      # Fixture teardown
      self.db.close()

  def test_isInt(self):
      ok_(foo.isInt(0))

  def test_dbConn(self):
      ok_(isinstance(self.db, DummyConnection))
```

Generative tests

- Each yield results in a test case

```
def test_generator():
    for i in xrange(0, 20, 2):
        yield check_even, i

def check_even(even_number):
    assert_true(foo.isEven(even_number))
    assert_false(foo.isEven(even_number + 1))
```

Attributes

- Add attribute tag to tests

```
from nose.plugins.attrib import attr

@attr('nothing')
def test_zero_equals_zero():
    assert 0 == 0
```

- Can set a specific value

```
@attr(speed=‘slow’)
```

Attributes

- Select attributes at runtime (-a/--attr)
  - nothing
  - speed=slow
- Python expression (-A/--eval-attr)
  - “not nothing”
  - “(speed==‘slow’ and not nothing)”

Skip tests

- Raise exception to report test skipped

```
from nose.plugins.skip import SkipTest

def test_skipme():
    raise SkipTest
```

Runs unittest tests

- Loads tests from unittest.TestCase subclasses
- Easily used as a front-end for legacy tests

Running tests

- Type `nosetests` at the top level
- Run a subset of tests
  - `cd package; nosetests`
  - `nosetests package/tests/module.py`
  - `nosetests package.tests.module:name`

Other features

- setuptools integration
- Plugin system
  - Debug, code coverage and profiling
  - Doctest runner
  - XUnit XML output

Pros

- No tedious mucking about with suites
- Can be used with legacy unittest tests
- Generative tests
- Plugins

Cons

- Not in standard library
- No official release supporting Python 3.x
  - py3k branch exists

py.test

- Written by Holger Krekel

Install py.test

- Actually part of pylib
- Download from http://pytest.org/
- Use easy_install:

```
easy_install py
```

Easy to get started..

- Just create a test file
  - Make sure it starts with ‘test_’
- Start adding test methods, or test classes:

```
import foo
import py.test

def test_isInt():
    pass
```

Uses ‘assert’

```
def test_isInt():
     assert True == foo.isInt(0)
     assert False == foo.isInt("not an int")


 def test_makeList():
     assert [1,2,3] == foo.makeList(1, 2, 3)
```

Checking exceptions...

- Use py.test.raises()

```
def test_divide():
    assert 0 == foo.divide(0, 1)

    # Dividing 1 by 0 should raise
    # ZeroDivisionError
    py.test.raises(
        ZeroDivisionError, foo.divide, 1, 0)
```

“Test” classes

Just start the class with “Test”:

```python
class TestFoo():
    def test_isInt(self):
        assert True == foo.isInt(0)
        assert False == foo.isInt("not an int")
```

Test Fixtures

- With test classes, use:

```python
def setup_method(self, method):
    # Fixture setup
    self.db = create_db_connection()

def teardown_method(self, method):
    # Fixture teardown
    self.db.close()
```

Funcargs

- Helps you to create arguments instead

```python
def pytest_funcarg__db(request):
    db = create_db_connection()
    request.addfinalizer(lambda: db.close())
    return db

def test_db(db):
    # ... do something …
    assert isinstance(db, DummyConnection)
```

- Funcargs are the preferred way in py.test

Funcargs

- Helps you to create arguments instead

```
def pytest_funcarg__db(request):
    db = create_db_connection()
    request.addfinalizer(lambda: db.close())
    return db

def test_db(db):
    # ... do something …
    assert isinstance(db, DummyConnection)
```

- Funcargs are the preferred way in py.test

Generative tests...

- Similar to nose...

```
def test_generative():
    for i in xrange(0,20,2):
        yield check_even, i

def check_even(even_number):
    assert True == foo.isEven(even_number)
    assert False == foo.isEven(even_number+1)
```

- This has been deprecated though

Generative tests...

- New style test generators
  - Still experimental
  - Encompasses several ways of doing:
    - Parameterized tests
    - Scenario tests

“Marking” tests...

- Can mark tests as “expected failure”

```
@py.test.mark.xfail
def test_xfail():
    assert 0 == 1
```

- Or with keywords

```
# A test marked with keyword "nothing"
@py.test.mark(nothing=True)
def test_zero_equals_zero():
    assert 0 == 0
```

“Marking” tests...

- Some keywords are added automatically
  - Filename
  - Class names
  - Function names

Skipping Tests

- Skip if an import fails

```
bogus = py.test.importorskip("bogus")
```

  - Skip all when done at module level
  - Skips an individual test when done inside a test function

- Skip for some other reason

```
py.test.skip("A short message")
```

Disabling a test class

- Great for disabling platform-specific tests:

```
class TestWin32Only:
    disabled = sys.platform != 'win32'
    # Test cases follow…
```

- Also good for disabling tests that require a specific software/hardware setup
    - Need a decent way to test for it
    - Keywords are useful for this too

Executing Tests

- Use the py.test command:

```
py.test [/path/to/file/or/dir] […]
```

- Automatically collects tests
- Begins executing tests immediately

Executing Tests

- Tests with a specific keyword:

```
py.test -k keyword
```

- Tests that don’t have a specific keyword:

```
py.test -k “-keyword”
```

Distributed Testing

- Two modes
  - Local
  - Remote

Locally

- Take advantage of multiprocessors machine
- Significant speedups if tests are IO bound
- To run tests in ‘<num>’ processes, use:

```
py.test -n <num>
```

Remotely

- Two strategies
  - Load balancing - every test one run once

    ```
    py.test --dist=load
    ```

    ```
    py.test -d
    ```

  - Multiplatform - run every test on each platform

    ```
    py.test --dist=each
    ```

Remote Mechanisms

- Use ssh

```
py.test -d --tx ssh=user@host//chdir=/tmp \
    --rsyncdir pkg
```

- Use a gateway

```
py.test -d --tx socket=ip:port//chdir=/tmp \
    --rsyncdir pkg
```

- Gateway launched via a small script
  - Does *not* require installing pylib

conftest.py

- Captures command-line args in a config file
- Great for storing remote test configuration

```
pytest_option_tx = ["ssh=localhost"]
pytest_option_dist="load"
# Paths are relative to conftest.py
rsyncdirs = [".", "../foo.py"]
```

- Can also configure plugin options

Other features

- Can drop into PDB on error
- Has a plugin system
  - Execute unittest-style TestCases
  - Coverage reporting (via figleaf)
  - Pylint integration
  - And many more

Pros

- Don’t need to collect the tests
- Generative tests
- Marking tests
- Distributed testing
- Excels with large test suites
- More to py.test than this small example

Cons

- Default output is a bit sloppy
- Funcargs and generative == breakage!
- No Python 3 support yet
- Distributed tests can hang on an internal failure

We have source!

http://www.szakmeister.net/misc/unit-testing.zip

Questions?