

Simplistix

Python as a Testing Tool

Chris Withers

Who am I?

- Chris Withers
- Independent Zope and Python Consultant
- Using Python since 1999
- Fan of XP

- What do I use Python for?
 - Content Management
 - Systems Integration
 - XML manipulation

“The Plan”

- Introduction to Testing
- Python's Unit Testing Framework
- Documentation Testing

Pre-requisites

- Knowledge of Python

Why Write Tests?

- Help define the problem you're solving
- Find unexpected breakage
- Automate a repetitive task
-
- Help with branching and merging

When to Test?

- Before any code is written
 - ideally...
- When you find a bug
- Make sure your test fails before it passes!

Types of Testing

- Functional testing
 - replay scenarios
 - good for customer acceptance
- Unit testing
 - each tests one behaviour of one function or method
- Documentation testing
 - makes sure your examples work
 - python specific

What to Unit Test?

- Smallest unit of functionality possible
- Why?
 - better chance you're testing what you think you're testing
 - less chance of two wrongs making a right
- Each type of behaviour
 - make sure edge cases are covered

The unittest module

- Python's unit testing framework
- Added in Python 2.1
- Based on PyUnit
 - which was based on JUnit

How to Write a Test

- The code we want to test

example1.py

```
def reverse(aList):  
    aList.reverse()  
    return aList
```

- The test

example1t.py

```
import unittest  
  
class ReverseTests(unittest.TestCase):  
  
    def test_normal(self):  
        # do import here, makes test independent  
        from example1 import reverse  
        # can use python's normal asserts  
        assert reverse([1,2,3])==[3,2,1]  
        # or more robust and informative unittest options  
        self.assertEqual(reverse([1,2,3]), [3,2,1])
```

How to Run a Test

- unittest provides several ways
- here's one...

```
import unittest

class ReverseTests(unittest.TestCase):

    def test_normal(self):
        # do import here, makes test independent
        from example1 import reverse
        # can use python's normal asserts
        assert reverse([1,2,3])==[3,2,1]
        # or more robust and informative unittest options
        self.assertEqual(reverse([1,2,3]), [3,2,1])

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

Helpful Methods

- `fail(message)`
- `assertEquals(x,y)`
- `failUnless(expression)`
- `failIfExpression`
- `assertRaises(Exception,callable,arg1,arg2,...)`

- Make sure it's clear why you've written the testing code that's there!

setUp and tearDown

- Unit Tests should be...
 - Atomic
 - Independent
 - Discrete
 - Concurrent
 - well maybe not...
- How do we do this?
 - start “from fresh” for each script
 - write tests carefully

setUp and tearDown

- Called for each test in a suite
- Should NOT fail
 - or consume resource if they do

example2t.py

```
class ReverseTests(unittest.TestCase):

    def setUp(self):
        from example2 import reverse
        self.reverse = reverse
        self.aList = [1,2,3]

    def tearDown(self):
        del self.aList

    def test_normal(self):
        self.assertEqual(self.reverse([1,2,3]), [3,2,1])

    def test_doesntMutate(self):
        self.assertEqual(self.reverse(self.aList), [3,2,1])
        self.assertEqual(self.aList, [1,2,3])
```

Good Testing Practice

- Watch your imports
 - They can fail!
 - Use factory methods

```
class MyTests(unittest.TestCase):  
  
    def _createThing(self, name):  
        from thing import Thing  
        return Thing(name)  
  
    def test_thing_name(self):  
        self.assertEqual(self._createThing('test').name, 'test')
```

Good Testing Practice

- Create base test classes
- Create re-usable test suites
 - can be used on multiple implementations

```
class baseTest(unittest.TestCase):  
  
    klass = NotImplemented  
  
    def _create(self.name):  
        return self.klass(name)  
  
    def test1(self):  
        o = self._create('test')  
        self.assertEqual(  
            o.getName(),  
            'test'  
        )  
  
    def test1(self): pass
```

```
from imp import Importer, Imp1, Imp2  
from base import baseTest  
  
class Imp1Test(baseTest): pass  
  
class Imp2Test(baseTest):  
  
    def testExtra(self):  
        o = self._create('test')  
        self.assertEqual(  
            o.getExtra(),  
            'TEST'  
        )
```

Good Testing Practice

- Destroy all fixtures
- Reset environment after each test

```
class TestDB(unittest.TestCase):  
  
    def setUp(self):  
        self._db = OpenDB()  
        self._db.beginTransaction()  
        self._db.clearAllTables() # known start state  
  
    def tearDown(self):  
        self._db.abortTransaction()  
        del self._db  
  
    def test_one(self):  
        # interesting discussion, what does this test:  
        self._db.insert('fish')  
        self.assertEqual(  
            self._db.select('animals from table'),  
            ['fish']  
        )
```

Limitations

- Test discovery
 - large multi-package applications
 - lots of tests
 - python's package infrastructure doesn't help!
- Hand-maintained script
 - test can get forgotten
- Heuristic discovery
 - can find things that aren't tests

Limitations

- Testing other languages
 - can't test for language specific problems
 - memory leakage
 - pointers out
 - don't get good feedback...
- Testing other frameworks
 - GUI's, etc
 - Should be handled by Functional Tests

DocTest

- Been around for a while
- Finally getting exposure
- Lets have a look at `example3.py`

The End

- Any questions?

Thankyou!

- Chris Withers
- chris@simplistix.co.uk
- <http://www.simplistix.co.uk>
- Do people want these slides to be available?