Testing

Amit Kotlovski
Open-Source Consultancy
http://amitkot.com

Agenda

- Testing with unittest
- Monkey patching

unittest

- An implementation of the JUnit family

Testing terminology

- Test fixture - required preparation for test
- Test case - the smallest unit of testing
- Test suite - a collection of test cases and test suites
- Test runner - a framework for running the tests

Code to be tested

```
class FunnyString(object):
    def __init__(self, string):
        self._data = string

    def __len__(self):
        return len(self._data)

    def __str__(self):
        return str(self._data)

    def __pos__(self):
        return self._data.upper()

    def __neg__(self):
        return self._data.lower()
```

Simple test

```
import unittest
from funnystring import FunnyString

class FunnyStringLengthTestCase(unittest.TestCase):
    def runTest(self): # the function containing the test
        fs = FunnyString('A biG Banana')
        self.assertEqual(len(fs), 12, 'incorrect length')

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

Run it

```
$ python testFunnyString.py
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

Let's add another test

```
import unittest
from funnystring import FunnyString

class FunnyStringLengthTestCase(unittest.TestCase):
    def runTest(self):
        fs = FunnyString('A biG Banana')
        self.assertEqual(len(fs), 12, 'incorrect length')

class FunnyStringStrTestCase(unittest.TestCase):
    def runTest(self):
        fs = FunnyString('A biG Banana')
        self.assertEqual(str(fs), 'A biG Banana',
                         'incorrect string representation')

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

Code duplication!

- The two test classes share most of the code
- Solution:
    1. Extract preparation code to `setUp()`
    2. Extract common base class

A common base class

```
class FunnyStringBaseTestCase(unittest.TestCase):
    def setUp(self):
        self.fs = FunnyString('A biG Banana')

    # Optional:
    def tearDown(self):
        pass

class FunnyStringLengthTestCase(FunnyStringBaseTestCase):
    def runTest(self):
        self.assertEqual(len(self.fs), 12,
                         'incorrect length')

class FunnyStringStrTestCase(FunnyStringBaseTestCase):
    def runTest(self):
       self.assertEqual(str(self.fs), 'A biG Banana',
                        'incorrect string representation')
```

Class duplication

- As we add more test each will require a separate but almost identical class
- Solution:
  - Multiple test methods share the same fixture
  - Test function begin with `test_`

Multiple tests in TestCase

```
class FunnyStringTestCase(unittest.TestCase):
    def setUp(self):
        self.fs = FunnyString('A biG Banana')

def test_length(self):
     self.assertEqual(len(self.fs), 12,
                      'incorrect length')

 def test_str(self):
    self.assertEqual(str(self.fs), 'A biG Banana',
                     'incorrect str representation')
```

Running the tests

- As a __main__ module
- From the command line

Inside the module

```
import unittest
from funnystring import FunnyString

class FunnyStringLengthTestCase(unittest.TestCase):
    def test_length(self):
        fs = FunnyString('A biG Banana')
        self.assertEqual(len(fs), 12,
                         'incorrect length')


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

Command line

```
# run all tests from testFunnyString5
$ python -m unittest --verbose testFunnyString5
test_int_exception (testFunnyString5.FunnyStringTestCase) ... ok
test_length (testFunnyString5.FunnyStringTestCase) ... ok
test_str (testFunnyString5.FunnyStringTestCase) ... ok

----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK

# run specific function test_length of the class FunnyStringTestCase
# in testFunnyString5
$ python -m unittest -v testFunnyString5.FunnyStringTestCase.test_length
test_length (testFunnyString5.FunnyStringTestCase) ... ok

----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

Assertions

- Validate expected conditions, e.g.:
  - **assertEqual**: actual value == expected value
  - **assertTrue**: received value is True
  - **assertRaises**: exception raised

Method	Checks that
`assertEqual(a, b)`	a == b
`assertNotEqual(a, b)`	a != b
`assertTrue(x)`	bool(x) is True
`assertFalse(x)`	bool(x) is False
`assertIs(a, b)`	a is b
`assertIsNot(a, b)`	a is not b
`assertIsNone(x)`	x is None
`assertIsNotNone(x)`	x is not None
`assertIn(a, b)`	a in b
`assertNotIn(a, b)`	a not in b
`assertIsInstance(a, b)`	isinstance(a, b)
`assertNotIsInstance(a, b)`	not isinstance(a, b)

Different assertions

```
class FunnyStringTestCase(unittest.TestCase):
    def setUp(self):
        self.fs = FunnyString('A biG Banana')

    def test_length(self):
        self.assertEqual(len(self.fs), 12, 'incorrect length')

    def test_str(self):
        self.assertTrue(str(self.fs) == 'A biG Banana',
                        'incorrect string representation')
```

Expected exceptions

```
>>> ', '.join([1, 2, 3])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: sequence item 0: expected str instance, int
found
```

Expected exeptions

```
# test_file.py
import unittest

class JoinTest(unittest.TestCase):
    def test_join_ints(self):
        with self.assertRaises(TypeError):
            ', '.join([1, 2, 3])

    def test_join_ints_check_message(self):
        with self.assertRaisesRegex(TypeError,
                                    '.*int found$'):
            ', '.join([1, 2, 3])
```

```
$ python3 -m unittest test_file
..
----------------------------------------------------------------------
Ran 2 tests in 0.000s

OK
```

setUp() and tearDown()

- Prepare the system for testing:
    - Create class instances, variables, etc.
    - Create tables in database and fill in data
- Clean up:
    - Reset database

Skipping tests

```
# test_file.py
import unittest

def my_upper(text):
    return text

class MyTestCase(unittest.TestCase):
   def test_my_upper(self):
        self.assertTrue(my_upper('eXampLe').isUpper())
```

Skipping tests

```
$ python -m unittest test_file
F
======================================================================
FAIL: test_my_upper (__main__.MyTestCase)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "<ipython-input-7-cc634f338464>", line 10, in test_my_upper
    self.assertTrue(my_upper('eXampLe').isupper())
AssertionError: False is not true

----------------------------------------------------------------------
Ran 1 test in 0.002s

FAILED (failures=1)
```

Skipping tests

```
# test_file.py
import unittest

def my_upper(text):
    return text

class MyTestCase(unittest.TestCase):

    @unittest.skip("demonstrating skipping")
    def test_my_upper(self):
        self.assertTrue(my_upper('eXampLe').isUpper())
```

```
$ python -m unittest test_file
s
----------------------------------------------------------------------
Ran 1 test in 0.005s

OK (skipped=1)
```

Q&A

https://secure.flickr.com/photos/21560098@N06/3836926854/

QUESTION
EVERYTHING

Thanks!

twitter @amitkot
www amitkot.com