Automated testing in Python

Poruri Sai Rahul,
Software Developer,
Enthought Inc.

Why?

- Why?
- What?
  - unittest
  - mock
- How?
  - unittest
  - mock

@rahulporuri

Why?

- Faster than manual testing.
  - Find bugs easily.
  - With help from Git, isolate exact set of changes that broke things.
- Understand current codebase.
- Understand test coverage.
- Using Continuous Integration, test on multiple platforms automatically.

What?

- Why?
- What?
  - unittest
  - mock
- How?
  - unittest
  - mock

What?

- Unittest
  - Makes writing and running tests easy
  - Available in the Python Standard Library.
- Mock
  - Allows user to mock unknowns for more concise tests.

How?

- Why?
- What?
  - unittest
  - mock
- How?
  - unittest
  - mock

How? - unittest

- Let’s start with something small - a function that takes two numbers and returns their sum

How? - using assert statements

```
sum.py
======
def sum(a,b):
    return a+b


test_sum.py
===========
assert sum(1,2) == 3
assert sum(3,4) == 7
```

How? - using assert statements and functions

```
sum.py
======
def sum(a,b):
    return a+b


test_sum.py
===========
def test_sum():
    assert sum(1,2) == 3
    assert sum(3,4) == 7


if __name__ == "__main__":
    test_sum()
```

What happens now?

```
sum.py
======
def sum(a,b):
    return a+b


test_sum.py
===========
def test_sum():
    assert sum('a',2) == ?
    assert sum(3,4) == 7


if __name__ == "__main__":
    test_sum()
```

How? - manually adding try, except blocks

```
sum.py
======
def sum(a,b):
    return a+b

test_sum.py
===========
def test_sum():
    try:
        assert sum('a',2) == ?
    except TypeError:
        print("summing {}, {} failed".format('a',2))


if __name__ == "__main__":
    test_sum()
```

What happens now?

```
sum.py
======
def sum(a,b):
    return a+b

test_sum.py
===========
def test_sum():
    assert sum('a','b') == ?
    assert sum([1,2], [3,4]) == ?


if __name__ == "__main__":
    test_sum()
```

How? - Writing your own Errors

```
sum.py
======
class InputError(Exception):
    def __str__(self):
        return “This function
                expects two int
                types as input”


def sum(a,b):
    if (not isinstance(a, int) or
        not isinstance(b, int)):
        Raise InputError
    else:
        return a+b
```

```
test_sum.py
===========
def test_sum():
    assert sum(‘a’,’b’) == ?
    assert sum([1,2], [3,4]) == ?


if __name__ == “__main__”:
    test_sum()
```

How? - using unittest to write tests

```
test_sum.py
===========
import unittest
from sum import sum


class TestSumFunction(unittest.TestCase):
    def test_should_work(self):
        assert sum(1,2) == 3
    def test_should_fail(self):
        assert sum([1,2], [3,4]) == [1,2,3,4]


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

How? - using unittest.TestCase methods

```
test_sum.py
===========
import unittest
from sum import sum, InputError


class TestSumFunction(unittest.TestCase):
    def test_should_work(self):
        self.assertEquals(sum(1,2), 3)
    def test_should_fail(self):
        with self.assertRaises(InputError):
            sum([1,2], [3,4])


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

How? - Explicitly loading test classes

```
test_sum.py
===========
import unittest
from sum import sum


class TestSumFunction(unittest.TestCase):
    def test_should_work(self):
        self.assertEquals(sum(1,2), 3)


if __name__ == “__main__”:
    suite = unittest.TestLoader().loadTestsFromTestCase(TestSumFunction)
    unittest.TextTestRunner(verbosity=2).run(suite)
```

How? - unittest (continued)

```
slope.py
===========
def calc_slope((x1, y1), (x2, y2)):
    slope = (y2-y1)/(x2-x1)
    return slope

test_slope.py
===========
import unittest
from slope import calc_slope


class TestSumFunction(unittest.TestCase):
    def test_should_work(self):
        self.assertEquals(calc_slope((1,2), (3,4)), 1)


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

How? - unittest and expected failures

```
test_slope.py
============
import unittest
from slope import calc_slope


class TestSumFunction(unittest.TestCase):
    def test_should_work(self):
        self.assertEquals(calc_slope((1,2), (3,4)), 1)
    @unittest.expectedFailure
    def test_should_also_work(self):
        self.assertEquals(calc_slope((0,1), (2,4)), 1.5)


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

How? - unittest (continued)

```
slope.py
===========
def calc_slope((x1, y1), (x2, y2)):
    slope = float(y2-y1)/(x2-x1)
    return slope

test_slope.py
===========
import unittest
from slope import calc_slope


class TestSumFunction(unittest.TestCase):
    def test_should_work(self):
        self.assertEquals(calc_slope((1,2), (3,5)), 1.5)


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

What happens now?

```
test_slope.py
============
import unittest
from slope import calc_slope


class TestSumFunction(unittest.TestCase):
    def test_should_work(self):
        self.assertEquals(calc_slope((1,2), (3,4)), 1)
    def test_should_also_work(self):
        self.assertEquals(calc_slope((0,1), (0,4)), ?)


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

How? - unittest (continued)

```
slope.py
===========
def calc_slope((x1, y1), (x2, y2)):
    if (x2-x1) != 0:
        slope = float(y2-y1)/(x2-x1)
    else:
        return numpy.inf
    return slope

test_slope.py
===========
import unittest
from slope import calc_slope

class TestSumFunction(unittest.TestCase):
    def test_should_now_work(self):
        self.assertEquals(calc_slope((0,2), (0,5)), numpy.inf)

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

- Why?
- What?
  - unittest
  - mock
- How?
  - unittest
  - mock

How? - unittest (continued)

Helpful commands

- python -m unittest -v test-file-name
- python -m unittest discover

- Why?
- What?
  - unittest
  - mock
- How?
  - unittest
  - mock

How? - mock

- Let’s start with something simple - mocking the sum function from earlier

How? - mocking an object

```
test_sum.py
===========
import unittest
import mock

import sum


class TestSumFunction(unittest.TestCase):
    @mock.patch('sum.sum')
    def test_should_work(self, mocked_sum):
        mocked_sum.return_value = 5
        self.assertEquals(sum.sum(1,2), 5)


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

How? - mocking an object (continued)

```
test_sum.py
===========
import unittest
import mock

import sum


class TestSumFunction(unittest.TestCase):
    @mock.patch('sum.sum')
    def test_should_work(self, mocked_sum):
        mocked_sum.return_value = 5
        self.assertEquals(sum.sum(1,2), 5)
        mocked_sum.assert_called_once_with(1,2)


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

How? - mocking an object's method

```
evolution.py
============
class Point(object):
    def __init__(self, x0, y0):
        self.x0 = x0
        self.y0 = y0

    def evolve(self):
        pass

    def result(self):
        return self.evolve()
```

```
test_evolution.py
=============
import unittest
import mock

from evolution import Point

class TestPoint(unittest.TestCase):
    @mock.patch.object(Point, 'evolve')
    def test_mock_method(self, mocked_evolve):
        mocked_evolve.return_value = 2
        p = Point(0,0)
        self.assertEquals(p.result(), 2)
        mocked_evolve.assert_called_once_with()
```

Credits

- Inspired by Ned Batchelder's Getting Started Testing talk at PyCon 2014