python™ 2 programming

Unit testing in Python

Unit testing in Python

- Contents
  - What is a Unit test
  - Testing with doctest
  - unittest framework
  - nose framework
- Summary

What is a Unit test

- A *unit test* is a piece of a code (usually a method) that invokes another piece of code and checks the correctness of some assumptions afterward.
- If the assumptions turn out to be wrong, the unit test has failed.
- *SUT* stands for system under test, or *CUT* (class under test or code under test).
  - When we test something, we refer to the thing we’re testing as the SUT/CUT.

Good unit test

- A unit test is almost always written using a unit-testing framework.
- It can be written easily and runs quickly.
- Anyone should be able to run it
- Characteristics:
  - Fully automated and repeatable
  - Trustworthy
  - Readable
  - Maintainable

When to write Unit tests?

- **Write code first , than write the tests for it.**
  - **Traditional way**
  - **The obvious way (?)**
- **Write tests first, and than write the production code**
  - **This approach is called TDD - Test Driven Development**
  - **NOTE:**
    - There are many different views on exactly what test-driven development means.
      In this course TDD means: test first development

Unit testing concepts

- Test fixture
  - A test fixture represents the preparation needed to perform one or more tests, and any associate cleanup actions.
    This may involve, for example, creating temporary or proxy databases, directories, or starting a server process.
- Test case
  - A test case is the smallest unit of testing. It checks for a specific response to a particular set of inputs.

Unit testing concepts (cont.)

- Test suite
  - A test suite is a collection of test cases, test suites, or both.
    It is used to aggregate tests that should be executed together.
- Test runner
  - A test runner is a component which orchestrates the execution of tests and provides the outcome to the user.
    The runner may use a graphical interface, a textual interface, or return a special value to indicate the results of executing the tests.

Testing with doctest

- **doctest** module searches interactive Python sessions text, and then executes those sessions to verify that they work exactly as shown.
- **Objectives**:
  - Check that a module’s docstrings are up-to-date by verifying that all interactive examples still work as documented.
  - Perform regression testing by verifying that interactive examples from a test file or a test object work as expected.
  - Write tutorial documentation for a package, liberally illustrated with input-output examples.

Testing a module

- Run a module as a main program

```
if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

- Testing is built into Python
  - Searches for docstrings containing interactive sessions

```
"""
    This is a sample module which
    does various date operations.

    >>> today = Date(13,12,1949)
    >>> print today
    13/12/1949
"""
```

- Run with the -v option

```
$ date.py -v
Trying:
    today = Date(13,12,1949)
Expecting nothing
ok
Trying:
    print today
Expecting:
    13/12/1949
ok
...
Test passed.
```

unittest framework

- Creating a Test case
  - Subclassing *unittest.TestCase*

```
import unittest
class TestStringMethods(unittest.TestCase):
  def test_upper(self):
     self.assertEqual('foo'.upper(), 'FOO')
  def test_isupper(self):
     self.assertTrue('FOO'.isupper())
     self.assertFalse('Foo'.isupper())
  def test_split(self):
    s = 'hello world‘
    self.assertEqual(s.split(), ['hello', 'world'])
    # check that s.split fails when the separator is not a string
    with self.assertRaises(TypeError):
      s.split(2)
if __name__ == '__main__':
    unittest.main()
```

Command-Line Interface

- **The unittest module can be used from the command line to run tests from modules, classes or even individual test methods**

```
python -m unittest test_module1 test_module2
python -m unittest test_module.TestClass
python -m unittest test_module.TestClass.test_method
python -m unittest discover
```

```
python -m unittest -v test_module
```

Test fixture

- **Class instances run one of the test_*()** methods, with self.widget created and destroyed separately for each instance

```
import unittest
class WidgetTestCase(unittest.TestCase):
    def setUp(self):
        self.widget = Widget('The widget')
    def tearDown(self):
        self.widget.dispose()
        self.widget = None
    def test_default_size(self):
        self.assertEqual(self.widget.size(),(50,50),'incorrect default size')
    def test_resize(self):
        self.widget.resize(100,150)
        self.assertEqual(self.widget.size(),(100,150),'wrong size after resize')
```

Create a Test suite

```
def suite():
    tests = ['test_default_size', 'test_resize']
    return unittest.TestSuite(map(WidgetTestCase, tests))
```

- **Run test suite**

```
unittest.TextTestRunner(verbosity=2).run(suite())
```

Asserts API

Method	Checks that	New in
`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`	2.7
`assertIsNot(a, b)`	`a is not b`	2.7
`assertIsNone(x)`	`x is None`	2.7
`assertIsNotNone(x)`	`x is not None`	2.7
`assertIn(a, b)`	`a in b`	2.7
`assertNotIn(a, b)`	`a not in b`	2.7
`assertIsInstance(a, b)`	`isinstance(a, b)`	2.7
`assertNotIsInstance(a, b)`	`not isinstance(a, b)`	2.7

nose test framework

- nose is an abstraction over unittests
- Based on naming conventions instead of writing special classes
- Use unittests as is and also to add some abstraction
- nose2 is support from python2.6 and above
  - Supports also tests parameters building

Naming conventions for nose:

- Each of the following test files will be run:
  - *test.py*
  - *test_views.py*
  - *test_models.py*
  - *testThingy.py*
- These files will not be run:
  - *not_a_test.py*
  - *myapp_test.py*
  - *some_test_file.py*
- Within test modules, nose will load tests from
  - *unittest.TestCase* subclasses
  - test functions (functions whose names begin with “test”)

Nose example

```
from unnecessary_math import multiply
def test_numbers_3_4():
    assert multiply(3,4) == 12
def test_strings_a_3():
    assert multiply('a',3) == 'aaa'
```

```
nosetests -v test_um_nose.py
```

```
@params((1, 2), (2, 3), (4, 5))
 def test_less_than(self, a, b):
         assert a < b
```

```
nose2 -v test_um_nose2.py
```

parameterized package

```
@parameterized([(1, 2, 3), (3, 4, 7)])
def test_f2_params(a, b, c):
    res = f2(a, b)
    assert res == c
```

```
class Test1(unittest.TestCase):
@parameterized.expand([(2, 3), (3, 4)])
def test_less_than(self, a, b):
          assert a < b
def test_upper(self):
        self.assertEqual('foo'.upper(), 'FOO')
class Test2(unittest.TestCase):
@parameterized.expand([("with:2,3",2, 3), ("with:3,4",3, 4)])
def test_less_than(self,_,a, b):
          assert a < b
def test_upper(self):
        self.assertEqual('foo'.upper(), 'FOO')
```