Python programming — Testing

Finn Årup Nielsen

DTU Compute
Technical University of Denmark

September 8, 2014

Overview

Testing frameworks: unittest, nose, py.test, doctest

Coverage

Testing of numerical computations

GUI testing

Web testing

Test-driven development

Testing frameworks

unittest: In the Python Standard Library, xUnit-style standard.

nose: Not in the stdlib. Simpler tests than `unittest`

py.test: Not in the Python Standard Library. Simpler tests than `unittest`. Better error messages than `nose`

doctest: In the Python Standard Library. Test specified in the documentation (i.e., in the docstring). Show to the user how to use the class and function

Testing with `nose`

Using `nose` with `mymodule.py` containing

```
def myfunction(x, y=2):
    return x+y

def test_myfunction():
    assert myfunction(1) == 3
    assert myfunction(1,3) == 4
```

Run the program `nosetests` (Campbell et al., 2009, p. 61–67)

```
$ nosetests mymodule.py
```

that discovers the `test_` functions. Or within Python:

```
>>> import nose, mymodule
>>> nose.run(mymodule)
```

doctest = documentation + testing . . .

`mymodule.py` with `myfunction` with Python code in the docstring:

```
def myfunction(x, y=2):
    """
    This function will add two numbers, e.g.,
    >>> myfunction(1,7)
    8

    The second argument is optional
    >>> myfunction(2)
    4
    """
    return x+y

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

...doctest

```
$ python mymodule.py
```

The body of the following conditional gets executed if the function is called as the main program (`__main__`), — rather than imported as a module:

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

`doctest.testmod()` will extract the code and the execution result from the docstrings (indicated with >>>), execute the extracted code and compare its result to the extracted result: Literal testing!

This scheme also works for errors and their messages.

Autodiscovery

Testing frameworks may have autodiscovery of testing functions. This are found with the `test_` prefix.

(Some have used `'__test__' in function_name` instead, see, e.g., (Martelli et al., 2005, section 8.8, "Running Unit Tests Most Simply"), but this format is apparently not recognized by nose and py.test)

Where to put the test? In the module with the implementation or in a separate module. There are conventions for the placement of testing code.

Coverage

Coverage = How much of the code has been executed

When testing: Coverage = How much of the code has be tested

There is a Python program for python code: coverage.py!

Example with a module `Etter2011Novo_nbc.py` containing `test_` functions and `nose.runmodule()`:

```
$ python-coverage run Etter2011Novo_nbc.py
$ python-coverage report | egrep "(Etter|Name)"
Name                                    Stmts   Miss  Cover
Etter2011Novo_nbc                         105      5    95%
```

Conclussion: Much but not all code tested here.

. . . Coverage

Consider `numerics.py` file:

```
def compute(x):
    """Add two to x."""
    if x == 2:
        return 5
    return x + 2

def test_compute():
    assert compute(1) == 3
    assert compute(-10.) == -8.

test_compute()
```

Then everything is ok ("collected 1 items") with:

```
$ py.test numerics.py
```

. . . Coverage

```
$ python-coverage run numerics.py
$ python-coverage report | grep numerics
numerics                                    8      1    88%
```

We haven’t tested all the code!

We haven’t testing the conditional for `x = 2`.

. . . Coverage

Lets add a new test to handle the conditional reaching 100% coverage in the test.

. . . Coverage

```
def compute(x):
    """Add two to x, except for 2."""
    if x == 2:
        return 5
    return x + 2

def test_divide():
    assert compute(1) == 3
    assert compute(-10.) == -8.
    assert compute(2) == 5           # new test!

test_divide()

$ python-coverage run numerics.py
$ python-coverage report | grep numerics
numerics                                      9       0    100%
```

Testing of numerical code

Consider a module `numerics.py` with:

```
def compute():
    return 1 - 1./3

def test_compute():
    assert compute() == 2./3
```

Using py.test:

```
$ py.test numerics.py
```

Results in an error(!?):

```
    def test_compute():
>       assert compute() == 2./3
E       assert 0.6666666666666667 == (2.0 / 3)
E        +  where 0.6666666666666667 = compute()
```

Testing of numerical code

The problem is numerical runoff:

$0.6666666666666667 \neq 0.6666666666666666$

New attempt:

```
def compute():
    return 1 - 1./3

def test_compute():
    assert abs(compute() - 2./3) < 0.0000000000000002
```

The testing does not fail now!

Testing of numerical code with Numpy

Now with a Numpy 2D array (a 4-by-4 matrix):

```
from numpy import *
def compute():
    return 1 - ones((4,4)) / 3
```

What now? Test all elements individually?

```
def test_compute():
    for row in compute():
        for element in row:
            assert abs(element - 2./3) < 0.0000000000000002
```

Perhaps we should also test that the size is appropriate?

Testing of numerical code with Numpy

```
from numpy.testing import *
def test_compute():
    assert_allclose(compute(), 2 * ones((4,4)) / 3)
```

`assert_allclose` is just one of several functions in `numpy.testing` that can be used to numerical testing.

Testing of numerical code with Numpy

We can still make the test fail with:

```
from numpy.testing import *
def test_compute():
    assert_equal(compute(), 2 * ones((4,4)) / 3)
```

Here `assert_equal` rather than `assert_close`, and this is what we get:

```
...
>                     raise AssertionError(msg)
E                     AssertionError:
E                     Arrays are not equal
E
E                     (mismatch 100.0%)
E                      x: array([[ 0.66666667,  0.66666667,  ...
E                            [ 0.66666667,  0.66666667,  0.6 ...
...
```

Zero-one-some testing . . .

Zero-one-some testing: If an instance (e.g., an input argument) can have multiple values then test with zero, one and multiple values. Here a `test_mean.py` module:

```
from numpy import mean, isnan

def test_mean():
    """Test the numpy.mean function."""
    assert isnan(mean([]))
    assert mean([5]) == 5
    assert mean([1, 0, 5, 4]) == 2.5
```

Fairly innocuous testing here. Mean of an empty list should/could be not-a-number. Mean of 5 should be 5 and mean of 1, 0, 5, 4 should be 2.5.

. . . Zero-one-some testing

Running the test. Oops:

```
$ python
>>> from test_mean import *
>>> test_mean()
/usr/local/lib/python2.7/dist-packages/numpy/core/_methods.py:57:
           RuntimeWarning: invalid value encountered in double_scalars
  ret = ret / float(rcount)
>>> test_mean()
>>> import numpy
>>> numpy.__version__
'1.7.1'
```

`mean([])` raise a warning (`exceptions.RuntimeWarning`) the first time you call it in Numpy 1.7 but not the second time!? `numpy.nan` is returned alright in both cases.

Test for different types . . .

```
from numpy import max, min

def mean_diff(a):
    """Compute mean of difference between consequetive numbers.

    Parameters
    ----------
    a : array_like

    """
    return float((max(a) - min(a)) / (len(a) - 1))

def test_mean_diff():
    assert mean_diff([1., 7., 3., 2., 5.]) == 1.5
    assert mean_diff([7, 3, 2, 1, 5]) == 1.5
```

. . . Test for different types

In this example there is a integer division problem (if we are in Python 2):

Runing `py.test` on the module gives:

```
    def test_mean_diff():
        assert mean_diff([1., 7., 3., 2., 5.]) == 1.5
>       assert mean_diff([7, 3, 2, 1, 5]) == 1.5
E       assert 1.0 == 1.5
E        +  where 1.0 = mean_diff([7, 3, 2, 1, 5])

typetesting.py:14: AssertionError
```

Note there are problem when `a` is of length zero and one.

GUI testing

See, e.g., Unit Testing with wxPython, where the test function sets up the wx frame but never calls the mainloop instead making a direct call to the callback method associated with a button.

Web testing

Selenium documentation

Modified from Getting started, testing one of my webservices:

```
from selenium import webdriver
from selenium.webdriver.common.keys import Keys

driver = webdriver.Firefox()
driver.get("http://neuro.compute.dtu.dk/cgi-bin/brede_bib_pmid2bwpaper")
assert "Brede" in driver.title
elem = driver.find_element_by_name("pmid")  # Find the 'pmid' form field
elem.clear()
elem.send_keys("16154272")                  # Query to the webscript
elem.send_keys(Keys.RETURN)

# An author by the name of Tadafumi should be somewhere in the text
assert driver.page_source.find("Tadafumi") != -1
```

httpbin

http://httpbin.org/ is a web service for testing an HTTP library:

```
>>> import requests

>>> user_agent = "fnielsenbot 0.1, http://www.compute.dtu.dk/~faan/"
>>> url = "http://httpbin.org/user-agent"
>>> requests.get(url, headers={"useragent": user_agent}).json()
{u'user-agent': u'python-requests/1.2.3 CPython/2.7.3 Linux/ ...

# Ups, wrong header.

# Trying with another:
>>> requests.get(url, headers={"user-agent": user_agent}).json()
{u'user-agent': u'fnielsenbot 0.1, http://www.compute.dtu.dk/~faan/'}
```

Checking code style and quality

Style checking in not testing per se but may help you catch errors and help you write beautiful code. Example:

```
$ pylint a_python_module.py
```

This use of `pylint` may result in:

```
************* Module a_python_module
W: 14: Unnecessary semicolon
C:  1: Missing docstring
W:  5:formatresult: Redefining name 'res' from outer scope (line 14)
C: 14: Invalid name "res" (should match (([A-Z_][A-Z0-9_]*)|(__.*__))$)
W: 17: No exception type(s) specified
C: 16: More than one statement on a single line
C: 17: More than one statement on a single line
W:  3: Unused import math
```

. . . Checking code style and quality

There are other tools for style checking: pep8. Example:

```
$ pep8 a_python_module.py
a_python_module.py:3:12: E401 multiple imports on one line
a_python_module.py:3:35: E262 inline comment should start with '# '
a_python_module.py:5:1: E302 expected 2 blank lines, found 1
a_python_module.py:11:53: W291 trailing whitespace
a_python_module.py:14:12: E702 multiple statements on one line (semicolon)
a_python_module.py:16:12: E701 multiple statements on one line (colon)
```

The tools are checking against Style Guide for Python Code, so-called PEP-8

Test-driven development

1. Write a small test for a basic functionality
2. Implement that functionality and no more
3. Run all tests
4. Write a small test that add an extra small functionality
5. Implement that functionality and no more
6. Run all tests
7. etc. . . .

More information

Style checking: A Beginner's Guide to Code Standards in Python - Pylint Tutorial

Dive Into Python, Chapter 13. Unit Testing. This covers only the `unittest` module.

Summary

Test your code!

Try using test-driven development.

py.test might be an ok framework. Nose also good.

Check your coverage. Ask why it is not 100%.

If variables with multiple values should be tested: Zero-one-some testing

Test for different variable types, e.g., it should work for both integers and floats.

It does not harm to use doctest, but it is probably difficult to test all you code with doctest, so use py.test & co.

Numpy has a separate testing framework. Good for Numpy computations.

References

Campbell, J., Gries, P., Montojo, J., and Wilson, G. (2009). *Practical Programming: An Introduction to Computer Science Using Python*. The Pragmatic Bookshelf, Raleigh.

Martelli, A., Ravenscroft, A. M., and Ascher, D., editors (2005). *Python Cookbook*. O'Reilly, Sebastopol, California, 2nd edition.