# 

# Python Testing Script

Mike Jackson, Steve Crouch, The Software Sustainability Institute

This work is licensed under the Creative Commons Attribution License

Copyright © Software Carpentry and The University of Edinburgh 2010-2013

See http://software-carpentry.org/license.html for more information.



**Things you should do are written in bold.**

Suggested dialog is in normal text.

Command-line excerpts and code fragments are in shaded fixed-width font.

## Prerequisites

Python, nose.

## Introduction

**Do Testing.ppt, up to the slide before What testing gives you slide.**

So how do we write tests?

Let’s take count\_records.py [this, along with sample text files is in testing-materials/]

import sys

# Given a file name, count the number of records

# in the file. Lines starting with "D" or "#"

# are ignored.

def count\_records(filename):

source = open(filename, 'r')

count = 0

# Count number of data records.

for line in source:

if line.startswith('#'): # Skip comments.

pass

elif line.startswith('D'): # Skip title line.

pass

else:

count += 1

source.close()

return count

if (len(sys.argv) < 2):

sys.exit("Missing file name")

filename = sys.argv[1]

print count\_records(filename)

How could I write tests? Well a very naïve way is to replace the last four lines with,

print count\_records(“empty.txt”)

print count\_records(“one.txt”)

print count\_records(“two.txt”)

print count\_records(“ten.txt”)

Then run it.

python count\_records.py

And visually inspect the results.

Note the first test. We should not just test for the expected or values we know work but for the unexpected e.g. empty lists, empty files, incorrect types, negative or out of bound values etc.

python count\_records.py

My tests are in the same file as my source code which isn’t very modular. So I’ll create a test\_count\_records.py file.

from count\_records import count\_records

print count\_records(“empty.txt”)

print count\_records(“one.txt”)

print count\_records(“two.txt”)

print count\_records(“ten.txt”)

But still this isn’t modular, so let’s define some test functions.

def test\_empty():

print count\_records(“empty.txt”)

def test\_one():

print count\_records(“one.txt”)

def test\_two():

print count\_records(“two.txt”)

def test\_ten():

print count\_records(“ten.txt”)

test\_empty()

test\_one()

test\_two()

test\_ten()

And make sure the four original prints have been removed, otherwise the output will be a bit overwhelming!

And let’s run it,

python test\_count\_records.py

But I still have to visually inspect the results to see if they’re right. So let’s add some validation.

def test\_empty():

if (0 != count\_records(“empty.txt”)):

print “FAIL”

def test\_one():

if (1 != count\_records(“one.txt”)):

print “FAIL”

def test\_two():

if (2 != count\_records(“two.txt”)):

print “FAIL”

def test\_ten():

if (10 != count\_records(“ten.txt”)):

print “FAIL”

And if we run that

python test\_count\_records.py

Fine. Now, to show we’re not cheating let’s hack our function to always return 5.

return 5

And run.

python test\_count\_records.py

And everything fails.

So let’s fix it and run it again to check we’ve fixed it…

python test\_count\_records.py

Fine. But we’re still having to write a lot of code to call our tests and check the results and report failures, and update our main function with each new test function, and we’re printing the output.

Why can’t the computer do this for us? It can!

We could write a shell script, but Python offers us something powerful. Nose is a Python testing library. It supports a nosetests command. It is an example of an xUnit test framework. You write test functions or classes and methods and it finds out what these are by their names. nosetests

* Looks for all files with test prefix.
* Looks for all functions with test prefix.
* Runs these functions.
* Prints a . for every test that passes.
* Prints a summary of the results.

So,

nosetests

runs our tests because our tests are in a test file which it looks for, and our test functions are prefixed by test, which it also looks for.

To show we’re not cheating we can remove our four calls the test functions and try again,

nosetests

And it still works as nosetests automatically searches for functions beginning with test to call.

nosetests can also handle our validation and reporting. We can replace our if-prints with asserts.

def test\_empty():

assert 0 == count\_records(“empty.txt”)

def test\_one():

assert 1 == count\_records(“one.txt”)

def test\_two():

assert 2 == count\_records(“two.txt”)

def test\_ten():

assert 10 == count\_records(“ten.txt”)

assert is traditionally expected, then actual. It takes a boolean and raises an error if the boolean is False.

And run again,

nosetests

And it still works.

If we re-introduce the bug, so our function always returns 5, and try again.

nosetests

It reports our failure!

nosetests has a lorra options e.g. select a specific Python module, class or function to test or test them all. For example:

nosetests test\_count\_records.py:test\_ten

It can also be hooked into test coverage and gives you control over how the results are logged and reported.

## Exercise

* **Look at utilities.py.**
* **Devise a set of at least five tests to test the calc\_mean() function**
* **Put them in test\_utilities.py**
* **Test the function with nosetests**
* **Think of useful and interesting test cases**

## Test results

Version control + automated tests such as nosetests allows for automated build and test.

An EPCC oncology project optimized and paralleled medical code. First they ran it to get the expected results, then set up an overnight test job to run the code and compare to the results. They could then optimize and parallelize in confidence.

Here is the VTK test dashboard, built using CDash.

**Browse to** [**http://open.cdash.org/index.php?project=VTK**](http://open.cdash.org/index.php?project=VTK)

Continuous integration tools detect version control commits, check out code, build, run tests, and publish, or run every few minutes and publish.

MICE’s MAUS test dashboard, built using Jenkins continuous integration server. MAUS tests are written in Python and run using nosetests.

**Browse to** [**https://micewww.pp.rl.ac.uk/tab/show/maus**](https://micewww.pp.rl.ac.uk/tab/show/maus)**.**

Jenkins will e-mail you when your job first fails and e-mail you again when it succeeds.

Faster you see a failure, faster you can fix it.

Public shame is a motivator too!

SSI guides about testing.

## How much testing is enough?

When to finish writing tests.

When it becomes not economic to do so in terms of time? Analogous to when to finish a proof reading a paper.

If you find bugs when you use your code, you did too little.

Learn by experience.

Note down how long it takes you, including interruptions and other work.

Tests, like code, should be reviewed.

Helps avoid tests that:

* Pass when they should fail.
* Fail when they should pass.
* Don't test anything. For example,

def test\_example():

# TODO: let’s finish this tomorrow

pass

Tomorrow never comes!

## Test driven development

Common to write code then write tests but there is an alternative…

Test first code second.

Red-green-refactor:

* Red - write tests based on requirements. They fail as there is no code!
* Green - write/modify code to get tests to pass.
* Refactor code - clean it up.