# 

# Python Testing Script

Mike Jackson, The Software Sustainability Institute

This work is licensed under the Creative Commons Attribution License

Copyright © Software Carpentry and The University of Edinburgh 2010-2012

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

**Try and leave 45m (or more if possible) for the exercise.**

A question for you…

* How many folk test their code?
* On many data sets?
* How many test it with incorrect inputs?
* Or boundary conditions?
* Or large amounts of data?

Testing gives us confidence that our software is

* Does what we want and expect it to.
* That it doesn’t behave unpredictably or mysteriously if given bad inputs or encounters errors.
* And that it behaves well if given large volumes of data, for example.

And a way to safely

* Refactor code
* Fix bugs. Nothing is worse than fixing a bug only to introduce a new bug.

And examples of how to use our code.

As bittermanandy says, “if it’s not tested, it’s broken” [bittermanandy, 10/09/2010]

So why isn’t testing done? Anyone?

Speak up if these sound familiar!

“I don’t write buggy code”, well, we are naturally very protective of our work and may refuse to accept that our code has bugs. Unfortunately, almost all code has bugs.

“It’s too hard”, but, if it’s hard to write a test for some code, then this is a good sign that the code is not well designed.

“It’s not interesting”, sometimes, testing is just viewed as not being interesting which means…

“It takes too much time and I’ve research to do”

And, this is a fair point. So, why should we care about testing?

Ariane 5, the successor to Ariane 4 had new and improved engines. Ariane 5 used Ariane 4 code. Unfortunately, the developers didn’t test the code properly. Ariane 5’s faster engines gave rise to a bug which caused a buffer overflow. And, the buffer overflow caused Ariane 5 to explode. So, some forgotten tests led to millions of pounds down the drain and some very red faces.

There is a story, possibly an urban myth, that during F16 test flights over Israel the navigation computer would crash and reboot. Seemingly one calculation included a division by altitude and when the plane flew over the Dead Sea…division by zero!

For us, consequences may not be so drastic but nevertheless they could be damaging. Consider Geoffrey Chang, a highly acclaimed researcher with the Scripps Institute, Beckman Award winner, designed to support researchers early in their academic careers, and with numerous publications to his name.

And then this publication, in Science…,

**Browse to** <http://www.sciencemag.org/content/314/5807/1875.2.long>

All because of a flipped sign!

And infamy lasts longer than 15 minutes,

**Browse to** <http://en.wikipedia.org/wiki/Geoffrey_Chang>

And he’s not alone.

“A Test of Corrections for Extraneous Signals in Gridded Surface Temperature Data”

R. McKitrick and P. Michaels, Climate Research 26(2) May 2004

**Browse to** <http://www.int-res.com/abstracts/cr/v26/n2/p159-173/>

Global warming is an artefact produced by weather stations in or near big cities, the ‘urban heat islands’ hypothesis.

But very quickly,

“McKitrick screws up yet again”, Tim Lambert’s blog, August 2004

**Browse to** <http://scienceblogs.com/deltoid/2004/08/26/mckitrick6/>

Note the graph. Their data had angles of latitude in degrees, a component they use expects angles in radians and they didn’t do a conversion!

And others took this up…

“McKitrick mucks it up”, John Quiggin’s blog, August, 2004

**Browse to** <http://crookedtimber.org/2004/08/25/mckitrick-mucks-it-up/>

And so in December,

“ERRATUM”, R. McKitrick and P. Michaels, Climate Research 27(3) 2004

**Browse to** <http://www.int-res.com/articles/cr2004/27/c027p265.pdf>

“formula for computing cosine of absolute latitude … takes the angle in radians, but our data were entered in degrees”

Do this too regularly and people may not trust our research, which could affect our chances for collaborations, publications or funding.

So how do we write tests?

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

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 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 main function and try again,

nosetests

And it still works.

It 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.

## 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!

## 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():

pass

## 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.

Suppose we have the following…

**Do Testing.ppt slide 1.**

**Ask students for test cases.**

Examples are,

* Rectangle A is to left of B, to right of B, above B, below B.
* Rectagle A is in B, B is in A, A and B are coincident.
* A and B overlap.
* A and B have single point overlaps on edges or corners.
* Edge cases
* Point cases

Note the conditions, no zero sized rectangles.

Is the specification wrong? Do we allow 0-area rectangles?

How to represent non-overlap?

Elaborate the specification - if no overlap then return "None".

Test driven development forces us to think about what the code should do, before we write it.

Clear up misunderstandings or gaps in any specification.

Code not easy to test is not easy to maintain.

**Open up new file, rectangle.py.**

**Add content**

# Given two rectangles return a rectangle representing where

# they overlap or None if they do not overlap.

# Each rectangle is represented as a list [x0, x1, y0, y1] where

# 0 < x0 < x1

# 0 < y0 < y1

def overlap(a, b):

return None

**Open up a new file, test\_rectangle.py**

**Add content**

from rectangle import overlap

def test\_example():

pass

**Run**

nosetests test\_rectangle.py

Let’s add tests for co-incident rectangles and enclosed rectangles and an overlap on the top right hand corner of A.

**Delete "test\_example" and add**

def test\_coincident():

a = [0, 2, 0, 2]

b = a

expected = a

actual = overlap(a, b)

assert expected == actual

Here is a test. We choose good function names so if the test fails I can easily understand what failed.

a and b are fixtures - what the test run on.

overlap is an action - what is done.

expected the expected result.

actual is the actual result.

assert takes a boolean and raises an error if the boolean is False. It gives a report on the success or failure of the test.

## Practical – write some tests first

Now your task is to come up with some more tests so in pairs discuss what tests you want to run and add these to test\_rectangle.py.

Just concentrate on writing your tests, don’t worry about the implementation of overlap. We’re doing test-driven development!

Use nosetests to check that the tests are syntactically valid. We expect all your tests to fail apart from test\_coincident, so don’t worry.

**Give the attendees 15 minutes to implement some tests.**

Here are mine, for an enclosed rectangle and an overlap on the top right hand corner of A.

def test\_a\_encloses\_b():

a = [0, 3, 0, 3]

b = [1, 2, 1, 2]

expected = b

actual = overlap(a, b)

assert expected == actual

def test\_a\_top\_right\_b():

a = [3, 6, 3, 6]

b = [0, 4, 0, 4]

expected = [3, 4, 3, 4]

actual = overlap(a, b)

assert expected == actual

def test\_a\_left\_of\_b():

a = [0, 2, 0, 2]

b = [3, 4, 3, 4]

expected = None

actual = overlap(a, b)

assert expected == actual

**Ask the students for others.**

Here we have tests for when there is overlap and one for where there is nothing expected.

**Run**

nosetests test\_rectangle.py

3 out of 4 fail.

Why does one pass? Because it's the one where None is expected.

We can now add code for co-incident rectangles

**Change "overlap" to be**

def overlap(a, b):

if (a == b):

return a

else:

return None

**Run**

nosetests test\_rectangle.py

## Practical – write the code to make the tests pass

Now, your task is, again in pairs, to write the code to make all your tests pass.

There are two solutions. A naïve solution involves a lot of ifs and conditionals (I did this when I first attended a boot camp).

But there is a more elegant alternative. This solution simplifies the problem and involves solving two smaller problems:

* Overlap on X axis.
* Overlap on Y axis.

Easier to determine overlap of two lines than two rectangles.

**Explain this using the images in Testing.ppt.**

And here’s some useful list things,

rectangle = [0, 2, 4, 6]

# List elements from 0..3

print rectangle[0] # rectangle[0] has value 0

print rectangle[1] # rectangle[1] has value 2

print rectangle[2] # rectangle[2] has value 4

print rectangle[3] # rectangle[3] has value 6

# Another approach

a, b, c, d = rectangle

print a # a has value 0

print b # b has value 2

print c # c has value 4

print d # d has value 4

Now go ahead and implement overlap so that your tests pass. You may want to focus on making one test pass at a time, or design a solution that makes them all pass.

**Keep 10 minutes for the end.**

**And one solution is…[sample solutions are in test-code/]**

**Add to rectangle.py**

def overlap\_axis(a0, a1, b0, b1):

start = max(a0, b0)

end = min(a1, b1)

if (end <= start):

return None

return [start, end]

If the line a0,a1 and b0,b1 overlap then this returns the overlapping part of the line.

Now we just get the overlap for both X and Y and if both are not None then we have the rectangle.

**Change "overlap" to be**

def overlap(a, b):

a\_x0, a\_x1, a\_y0, a\_y1 = a

b\_x0, b\_x1, b\_y0, b\_y1 = b

overlap\_x = overlap\_axis(a\_x0, a\_x1, b\_x0, b\_x1)

overlap\_y = overlap\_axis(a\_y0, a\_y1, b\_y0, b\_y1)

if (overlap\_x == None) or (overlap\_y == None):

return None

return overlap\_x + overlap\_y