Python Testing Script

Mike Jackson, The Software Sustainability Institute

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) 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.

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.

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…
[http://www.sciencemag.org/content/314/5807/1875.2.long]
All because of a flipped sign!
And infamy lasts longer than 15 minutes…
[http://en.wikipedia.org/wiki/Geoffrey_Chang]

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

Testing gives us

- Confidence that your code does what it is supposed to.
- Examples of what the code is supposed to do - a runnable specification.
- Ability to detect, and fix, bugs more quickly.
- Confidence to refactor or fix bugs without creating new bugs.
- Examples of how to use your code.

So how do we write tests?

Let’s take count_records.py,

```
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 would 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, testing a boundary condition. 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_copunt_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.
assert 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.

VTK test dashboard, built using CDash.
Browse to 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.
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!

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.

Scenario

Suppose we have the following:
Write this on a whiteboard or in a window:

INPUT: two rectangles.
OUTPUT: rectangle representing overlap between the two input rectangles.
Rectangle is defined as a list of four points:
  [x0, x1, y0, y1]
where:
  0 < x0 < x1
  0 < y0 < y1
FUNCTION: rectangle overlap(rectangle1, rectangle2)

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.

Python tests

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
nosetests test_rectangle.py:test_example
```

An xUnit test framework, from an idea by Kent Beck. CUnit, JUnit etc.

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

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
```

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.
assert gives a report on the success or failur eof the test.
Have 4 tests for when there is overlap and one for where there is not.
Test function names are chosen to be meaningful so we know what failed.
Run

```
nosetests test_rectangle.py
```

3 out of 4 fail.
Why does one pass? Because it's the one where None is expected.
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
```

2 out of 4 fail.

Exercise

In pairs - pair programming - write test_rectangle.py to implement your tests.
Write the code in rectangle.py to make the tests pass.

Wave if you get stuck and a helper will come round!

How did you get on?

Solution involves solving two problems:

- Overlap on X axis.
- Overlap on Y axis.

Easier to determine overlap of two lines so let’s write a function.

Explain this using the images in Testing.ppt

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
```

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
```