# Python Testing Script

**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.

## Introduction

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

Common to write code then write tests.

But people don't write tests.

Why?

* People don't write buggy code.
* It runs fine on a couple of examples.
* It looks like it's OK.
* There's no time, there's a conference deadline.

What testing gives you:

* 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.

## Test driven development

Test driven development.

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.

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.

## 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):

pass

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

**Add content**

from rectangle import overlap

def test\_example():

pass

Nose is a Python testing library. It supports a nosetests command.

**Run**

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.

**Run**

nosetests test\_rectangle.py

nosetests test\_rectangle.py:test\_example

Let’s aadd 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

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.

**Can sketch these on a whiteboard if desired.**

* a0---a1 b0---b1 None
* b0---b1 a0---a1 None
* a0---b0---a1---b1 b0,a1
* b0---a0---b1---a1 a0,b1
* b0---a0---a1---b1 a0,a1
* a0---b0---b1---a1 b0,b1

**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.

## Test results

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

Publish results.

OGSA-DAI dashboard, custom built in Ruby and using ANT build tool and JUnit Java unit test framework.

**Browse to** [**http://daiserver.epcc.ed.ac.uk/~ogsadai-tests/**](http://daiserver.epcc.ed.ac.uk/~ogsadai-tests/) **and click on a link.**

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.

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!

## Fixtures and decorators (if time)

What if we had a lot of work to do to set up a fixture?

**Create test\_example.py and add content**

import sys

def test\_one():

print >> sys.stderr, 'test\_one'

rectangle = [0, 5, 1, 5]

assert rectangle[0] == 0

def test\_two():

print >> sys.stderr, 'test\_two'

rectangle = [0, 5, 1, 5]

assert rectangle[1] == 5

def test\_three():

print >> sys.stderr, 'test\_three'

rectangle = [0, 5, 1, 5]

assert rectangle[2] == 1

**Run**

nosetests test\_example.py

May end up with duplicated code.

Could pull out into a custom method.

**Change test\_example.py content to**

import sys

rectangle = []

def create\_rectangle():

global rectangle

rectangle = [0, 5, 1, 5]

print >> sys.stderr, 'create\_rectangle'

def test\_one():

print >> sys.stderr, 'test\_one'

create\_rectangle()

assert rectangle[0] == 0

def test\_two():

print >> sys.stderr, 'test\_two'

create\_rectangle()

assert rectangle[1] == 5

def test\_three():

print >> sys.stderr, 'test\_three'

create\_rectangle()

assert rectangle[2] == 1

We need “global rectangle” in create\_rectangle else Python will just create a new local variable in the assignment.

**Run**

nosetests test\_example.py

But we still have the duplicated code.

nose supports a "setup" function to create a fixture.

**Change test\_example.py content to**

import sys

rectangle = []

def setup():

global rectangle

rectangle = [0, 5, 1, 5]

print >> sys.stderr, 'setup'

def test\_one():

print >> sys.stderr, 'test\_one'

assert rectangle[0] == 0

def test\_two():

print >> sys.stderr, 'test\_two'

assert rectangle[1] == 5

def test\_three():

print >> sys.stderr, 'test\_three'

assert rectangle[2] == 1

We could just hard-code the value of rectangle but this doesn’t generalise to more complex set-up scenarios e.g. those where our fixture(s) are populated using helper functions.

**Run**

nosetests test\_example.py

Note how setup is called before the tests are run.

The test framework removes the need for duplicated code.

But, supposing a test changed the fixture?

**Change test\_example.py content to**

import sys

rectangle = []

def setup():

global rectangle

rectangle = [0, 5, 1, 5]

print >> sys.stderr, 'setup'

def test\_one():

print >> sys.stderr, 'test\_one'

assert rectangle[0] == 0

def test\_two():

print >> sys.stderr, 'test\_two'

rectangle[0] = 10

rectangle[1] = 10

rectangle[2] = 10

assert rectangle[1] == 10

def test\_three():

print >> sys.stderr, 'test\_three'

assert rectangle[2] == 1

**Run**

nosetests test\_example.py

The tests can fail since there is no guarantee of the order in which they are run.

But nosetests saves us again.

We can request that the setup be run before every test.

**Change test\_example.py content to**

import sys

from nose import with\_setup

rectangle = []

def setup\_each():

rectangle = [0, 5, 1, 5]

print >> sys.stderr, 'setup'

@with\_setup(setup\_each):

def test\_one():

print >> sys.stderr, 'test\_one'

assert rectangle[0] == 0

@with\_setup(setup\_each):

def test\_two():

print >> sys.stderr, 'test\_two'

rectangle[0] = 10

rectangle[1] = 10

rectangle[2] = 10

assert rectangle[1] == 10

@with\_setup(setup\_each):

def test\_three():

print >> sys.stderr, 'test\_three'

assert rectangle[2] == 1

**Run**

nosetests test\_example.py

Note how the setup is now called before every test.

@with\_setup is a Python decorator. Basically, it tells nose to run setup\_each before each test function.

This does mean that tests take a bit longer to run since the setup is done for every test.

But remember, your time is far more valuable than a computer's time!