# Advanced Topics: Biopython

## Day Three – Testing

Peter J. A. Cock

The James Hutton Institute, Invergowrie, Dundee, DD2 5DA, Scotland, UK

23rd – 25th January 2012,
Workshop on Genomics, Český Krumlov, Czech Republic


# Talk Outline

# Why test your code?

- To make sure it does what you want it do do!
- Later on, to make sure it still does what you want it do do (e.g. after updating a library)
- To check changes don't have unexpected side effects

# But writing tests wastes time!

- Yes, writing tests takes time.
  Sometimes as much as writing the code.
- BUT, overall they should save you time.
  Especially if you are doing experimental work based on it.
- PhD students may have to convince their supervisor . . .

# Writing good tests is hard!

- Simple tests check a typical case of good input data
  That is important.
- You must also test with bad input data
  Error handling is *very* important.

# Test granularity

- Big tests might verify the output of a lot of code in one go
  But if it fails, where is the error?
- Many little tests can verify each function individually
  Cause of a failure is usually clear
  Can be used with test driven development
- (Aside: Big functions are a bad idea - modularise!)

# Testing frameworks in Python

There are two main test frameworks included in Python itself,

- unittest - You write test objects with methods that each run a test
  `http://docs.python.org/library/unittest.html`
- doctest - You write examples within docstring documentation comments
  `http://docs.python.org/library/doctest.html`

Also the `assert` statement is useful.

# Example: mycode.py

Suppose you have module `mycode`, with a function `power`:

```
def power(value, exponent):
    answer = 1
    for i in range(exponent):
        answer *= value
    return answer
```

It needs some tests, but also some documentation...

# Example with assert

```
def power(value, exponent):
    """Calculate value^exponent and return it.
    """
    answer = 1
    for i in range(exponent):
        answer *= value
    return answer

assert power(5, 3) == 125
```

# Built in documentation

That triple quoted string at the start of the function is the function's *docstring*, accessed via `help(...)` command:

```
>>> from mycode import power
>>> help(power)

Help on function power in module mycode:

power(value, exponent)
    Calculate value^exponent and return it.
```

# Example with doctest

```
def power(value, exponent):
    """Calculate value^exponent and return it.

    Example:
        >>> power(5, 3)
        125
    """
    answer = 1
    for i in range(exponent):
        answer *= value
    return answer

import doctest
doctest.testmod(verbose = True)
```

# Example with doctest

Recall that triple quoted string at the start of the function is the function's *docstring*, accessed via the `help(...)` command:

```
>>> from mycode import power
>>> help(power)
Help on function power in module mycode:


power(value, exponent)
    Calculate value^exponent and return it.

    Example:
        >>> power(5, 3)
        125
```

# Example with doctest

The *docstring* can contain *doctest* strings, real examples of commands and their output like:

```
>>> power(5, 3)
125
```

Using *doctest* these examples are run and compared to the expected output.

# Limitations with doctest

- The output is compared as a string - cross platform differences can be a problem
- Examples of failures don't usually make good documentation.
- Think of *doctest* strings as documentation examples that gets tested

# Example with unittest

```
import unittest
from mycode import power

class TestPower(unittest.TestCase):
    #Start tests with special method name test_...
    def test_squared( self ):
        for x in [1, 2, 10, 12345]:
            self.assertEqual(x*x, power(x,2))

runner = unittest.TextTestRunner(verbosity = 2)
unittest.main(testRunner=runner, exit=False)
```

# Comments on unittest

- You can have lots of test methods
- You can easily test exceptions, see `assertRaises`
- You can even generate test methods dynamically
- i.e. Very flexible

# Unit testing in Biopython

- Main modules try to include some `doctest` examples
- Most new tests use the `unittest` framework
- Some old test scripts work with an expected output file
- We also have `doctest` examples in the main tutorial

# Nightly testing with Buildbot

- Some bugs are platform or Python version specific
  $\rightarrow$ running the tests on one machine often not enough
- We use BuildBot – `http://trac.buildbot.net/`
  See `http://testing.open-bio.org/biopython/`
- Runs tests every night on Linux, Mac OS X, Windows, covering Python 2.5, 2.6, 2.7, 3.1, 3.2, and Jython
- Some projects run their tests for every commit!

# Testing in your code

- My own code has improved since I started writing tests
- Unit testing is woefully underused in Bioinformatics
- Don't trust non-trivial programs without unit tests
- Please do write tests for your own code!

Even standalone single file Python scripts can use `doctest` and `unittest` – so that's not an excuse.

# Testing in your code

Write some tests for this power function:

```
def power(value, exponent):
    answer = 1
    for i in range(exponent):
        answer *= value
    return answer
```

Perhaps compare it to Python's built in power operator,

```
>>> print 5**3
125
```