Writing Tests for Biopython modules

Brad Chapman (chapmanb@uga.edu)

November 7, 2008

Contents

1 Purpose and Justifications

So you want to write a Test for a Biopython module? Great! Providing comprehensive tests for modules is one of the most important parts of keeping code up to date and working the way you expect it to. It also tends to be one of the most undervalued aspects of contributing, so this document is designed to make writing good test code as easy as possible.

We start off with the simple assumption that there is a module you wrote (or which doesn't already have tests), and you want to test it out. We'll call it `MyModule`, for lack of a better name, from now on.

2 Understanding the Biopython test system

Biopython tests are found in `biopython/Tests` and each test will have two important files and directories involved with it:

1. `test_MyModule.py` – The actual test code for your module.
2. `MyModule` – A directory where any necessary input files will be located. Any output files that will be generated should also be written here (and preferrably cleaned up after the tests are done) to prevent clogging up the main Tests directory.

The main engine that runs all the tests is `run_tests.py`. If you name your file with a `test_` prefix and put it in the `Tests` directory, `run_tests.py` will find it and run it.

3 Writing the Tests

Okay, now that we're oriented with the important parts of the test framework, we'll get down to writing tests. We will use the unittest framework, included with Python since version 2.1, and documented in the python Library Reference (which I know you are keeping under your pillow, as recommended).

To get started, here's a framework to copy and paste:

```
import os
import sys
import unittest

def run_tests(argv):
    test_suite = testing_suite()
    runner = unittest.TextTestRunner(sys.stdout, verbosity = 2)
    runner.run(test_suite)

def testing_suite():
    """Generate the suite of tests.
    """
    test_suite = unittest.TestSuite()

    test_loader = unittest.TestLoader()
    test_loader.testMethodPrefix = 't_'
    tests = [MyModuleTestOne]

    for test in tests:
        cur_suite = test_loader.loadTestsFromTestCase(test)
        test_suite.addTest(cur_suite)

    return test_suite

class MyModuleTestOne(unittest.TestCase):
    def setUp(self):
        pass

    def tearDown(self):
        pass

if __name__ == "__main__":
    sys.exit(run_tests(sys.argv))
```

The two functions `run_tests` and `testing_suite` just do all of the unittest setup – so you don't really need to understand everything about the framework to write the tests. All you'll need to do is write classes, like MyModuleTestOne, and add them to the lists of test classes in the `testing_suite` function and you'll be set.

Here's an example of a Test class, which will lead into a description of the different parts which go into making one:

```
class MyModuleTestOne(unittest.TestCase):
    def setUp(self):
        self.handle = open("MyModule/input_file.txt")
        self.output_file = "MyModule/output.txt"

    def tearDown(self):
        self.handle.close()
        if os.path.exists(self.output_file):
            os.remove(self.output_file)

    def t_simple_parsing(self):
```

```
        """Test to be sure that MyModule can parse input files.
        """
        parser = MyModule.RecordParser()
        rec = parser.parse(self.handle)
        assert rec.id = "TheExpectedID"

    def t_output(self):
        """Ensure that we can write proper output files.
        """
        parser = MyModule.RecordParser()
        rec = parser.parse(self.handle)
        output_handle = open(self.output_file, "w")
        rec.write_to_file(output_handle)
        output_handle.close()
```

We'll cover the important parts of this class one at a time:

- It should derive from `unittest.TestCase` and should cover one basic aspect of your code (parsing into Record objects, parsing into Sequence objects...)
- `setUp` – code that you want to run before running each test in the class. This might set up expected files or open files – it just prevents having to rewrite the same code over and over in each test.
- `tearDown` – code to clean up after your test is done. This will close up handles you were using and clean up produced files that you don't need any longer.
- The tests are prefixed with `t_` and each test should cover one specific part of what you are trying to test. You can have as many tests as you want in a class.
- The documentation strings of the tests are used when running them – so write something useful that will provide help if a test is failing.
- You should test expected parts of your output using standard python assertions. This makes explicit the things you think are important enough to remain the same over multiple runs of the tests and revisions of the code base.

When you are done, the tests should run happily by doing `python test_MyModule.py` and you should expect output like:

```
Test to be sure that MyModule can parse input files. ... ok
Ensure that we can write proper output files. ... ok

----------------------------------------------------------------------
Ran 2 tests in 0.225s
```

You should continue like this until you feel you've suitable tested the parts of the code so that someone running this test later on can tell if they've broken anything essential. You have to draw a line between testing everything in the world, and getting tests written in a reasonable amount of time. Tests are important, but you'd much rather be writing code, right? So get them done and you'll be happy.

4 Getting the test integrated in the testing framework

Well, you've written all your tests and they all pass and your are feeling very happy. There is just one last step to get things integrated with Biopython. Every time someone runs `python run_tests.py` each test is

executed, and then checked against the expected output (all 'ok's) to make sure nothing has broken. Well, you need to make sure the tests knows what the expected output is.

To do this takes just a second:

```
python run_tests.py -g test_MyModule.py
```

This will run your test, and write the output to `output/test_MyModule`. You should go in and check this file to make sure everything is as expected. Then, you're all set.

So, now you should be able to run all of the Biopython tests, see your test show up, and get a happy ok from the testing framework.

Good job – now you either need to check your tests in if you have CVS access, or just send your `test_MyModule.py` and contents of `MyModule` to someone with CVS priviledges.

Thanks for the tests.