Unit Testing

based on "Software Carpentry for Scientists"

swc.scipy.org

Niko Wilbert

April 24, 2009

What is Unit Testing?

- Write tests for the smallest "units" of your program (but "smallest" does not mean small in an absolute sense).
- This tells you precisely where the bugs are.
- Combine the tests into a test suite to run them automatically.
- Ideally cover all possible cases (so new program parts will not run into unknown bugs).
- Can instantly check if code changes introduced new errors or regressions (of course this requires good test coverage).

- Culminates in "test driven development".

JUnit and Its Children

- JUnit is a testing framework originally written by Kent Beck and Erich Gamma in 1997.
  - Made testing easy enough that programmers actually started doing it.
  - Now integrated into almost all Java IDEs.
- Widely imitated:
  - Workalikes are available C++, Perl, .NET, etc.
  - Once you know one, you can easily learn and use the others.
  - Add-ons for measuring test execution times, recording tests, testing web applications, etc..
- We look at Python's version, called unittest.

The Basics

- Define one method for each test.
  - Method name must begin with test.
  - Method must not take any parameters (other than self).
  - Shouldn't return anything.
- Group related tests together in classes.
  - Which must be derived from unittest.TestCase.
- Call unittest.main(), which:
  - Searches the module (i.e., the file) to find all classes derived from unittest.TestCase.
  - Runs methods whose names begin with test in an arbitrary order. Another reason not to make tests dependent on each other.
  - Counts and reports the passes, fails, and errors.

Example

```
import unittest

class TestAddition(unittest.TestCase):

    def test_zeroes(self):
        self.assertEqual(0 + 0, 0)
        self.assertEqual(5 + 0, 5)
        self.assertEqual(0 + 13.2, 13.2)

    def test_positive(self):
        self.assertEqual(123 + 456, 579)
        self.assertEqual(1.2e20 + 3.4e20, 3.5e20)

    def test_mixed(self):
        self.assertEqual(-19 + 20, 1)
        self.assertEqual(999 + -1, 998)
        self.assertEqual(-300.1 + -400.2, -700.3)

if __name__ == '__main__':
    unittest.main()
```

Checking

Checking methods include:

- assert_(condition): check that something is true (note the underscore)
- assertEqual(a, b): check that two things are equal
- assertNotEqual(a, b): the reverse of the above
- assertRaises(exception, func, args): call func with arguments (if provided), and check that it raises the right exception
- fail(): signal an unconditional failure

What Tests To Write First

- Tests you expect to succeed.
  - Simplest interesting case (e.g., sort a list of two values).
  - General case (e.g., sort a list of nine values).
  - Boundary cases (e.g., sort the empty list, or a list of one value).
  - Special cases that might cause problems (like duplicate elements in a list).
- Tests you expect to fail (e.g. is the proper exception raised?).
- Make sure that the tests match the specifications given in the docstrings (in some ways unittests themselves are specifications).
- Sanity tests.
  - Make sure data structures are not corrupted.
  - If there is redundant information, check consistency.

More unittest Techniques

- If the test class defines a setUp method (fixture), unittest calls it before running each test. And if there's a tearDown method, it is run after each test.
- Can use TestSuite objects to combine tests from multiple test modules:

```
suite = unittest.TestSuite()
suite.addTest(unittest.makeSuite(MyTestCase))
...
unittest.TextTestRunner().run(suite)
```

- Input and output often seem hard to test
  - Store a bunch of input files in a subdirectory?
  - Create temporary files when tests are run?
  - The best answer is to imitate I/O using strings
  - Python's StringIO and cStringIO modules can read and write strings instead of files

Usage

- Isn't all this too much work?
  - Only takes a few minutes, but can save you hours (weeks, month...).
  - The time savings rise exponentially with project size and complexity.
  - Your programs always contain more errors than you think.
  - You either learn this the hard way or the smart way.
- Each time you find a new bug, write a corresponding unit test.
  - The test is a safeguard against regression.
  - It was a bug, so the error is not obvious. Add tests for this type of problem in similar programm parts.
  - First attempts to fix bugs are often wrong (or the solution introduces new bugs), make sure you really fix it.
- Nothing more embarrassing than publishing false results based on a programming error.