Unit Testing and Continuous Integration

Jim Chiang

2016-10-07

Outline

- Why test code?
- What is unit testing?
- Benefits of unit testing
- Writing unit tests
- Recommended practices
- Unit testing frameworks
- unittest example
- Continuous Integration
- Using Travis-CI
- Some resources on unit testing

Why Test Code?

- We all test code: Any time code is executed and we do something with the output, we are testing it at some level.
- Any code base of significant size contains bugs.
- Any new development can introduce bugs or break things in other parts of the code.
- A comprehensive set of tests ensures, in a systematic way, that the code continues to perform as expected if any new development occurs.

What is unit testing?

There are many of types of software testing: system, integration, regression, functional, black box, white box, regression, etc.. These many of these overlap in purpose and are combined in application.
For this talk, we will concentrate on unit testing. This is the most basic kind of testing. It exercises the lowest levels of functionality: functions and class methods, or even single lines of code, and therefore affects most directly the process of developing software as it is occurring.

Here is a simple example:

```
def test_ISOT_to_MJD(self):
    isot = '2016-10-27T00:00:00.000'
    self.assertEqual(ISOT_to_MJD(isot), 57688.)
```

Benefits of unit testing

- Since the testing is finely grained, many bugs can be localized very quickly and precisely.
- The tests themselves serve as interface documentation since they provide canonical usage examples that are guaranteed to work.
- Comprehensive, quickly executing tests allow one to "refactor" aggressively since any new bugs will be found right away. Refactoring is the practice of improving the design of existing code while keeping its functionality the same. Unit tests help ensure that outcome.

Writing unit tests

- Tests should run quickly and automatically. One should feel free to run unit tests many times during a session.
- Tests should be self-checking. Only failures should produce output that needs to be parsed.
- Test code should be standalone and self-contained. Dependencies on external data (which may have limited accessibility or which may change over time, e.g., the contents of a web page) should be kept to a minumum.
- Things to test:
  - Test for success: Given specific inputs, does the function or method produce the expected outputs?
  - Test for failure: For bad input, does the function or method fail in the expected way? e.g., by raising specific exceptions or returning specific error codes.

```
def test_ISOT_to_MJD_bad_input(self):
    "Pass the correct type (str), but with an invalid value."
    isot = '10-27-2016 00:00:00.000'
    self.assertRaises(ValueError, ISOT_to_MJD, isot)
```

Some recommended practices

- When dealing with a new bug, first write a unit test that exposes and isolates it, and add that test to the test suite. As soon as the test passes, the bug is fixed.
- Before adding new features, write the unit test code first. The tests should fail at the outset, before any work on the production code has been done. Once the tests pass, the development is done. This is known as Test Driven Development. This
  - forces the developer to concentrate on interfaces rather than the underlying implementation,
  - helps ensure that only the needed code is added, and thereby reduces the likelihood of introducing new bugs via untested features.

More recommended practices

- Make sure the full set of tests in a package or repo all pass before pushing your code to the master branch of the main repository, otherwise you may break the build for others. Use of branches and continuous integration services can help prevent this.
- Coverage: This is the fraction of code that is tested by the full test suite. 100% is "ideal", but with limited resources it is better to concentrate on testing the parts of the code that are riskiest in the sense that the cost of failure is the greatest.

Unit testing frameworks

Unit test frameworks provide tools to help you write, organize, and execute your tests.

- Runs all tests and aggregates results
- Only reports failures (with details)
- Provides assertion methods for different types of tests
- Enables use of test "fixtures" which allow one to define precisely the environment or context in which the test runs.
- Allows one to define "test suites" that can be run to test related but otherwise isolated parts of the code.
- For python, unittest (and its xunit cousins), py.test, numpy.testing,

unittest example

I have a toy package, dateconvert, on GitHub.

```
import unittest
import numpy as np
import dateconvert

class DateConversionTestCase(unittest.TestCase):
    "Test case for date conversion utility"
    def setUp(self):
        "Read in some validated MJD-ISOT pairs."
        self.test_data = np.recfromtxt('mjd_isot_test_values.txt')

    def tearDown(self):
        "Nothing to tear down."
        pass

    def test_MJD_to_ISOT(self):
        "Test conversion from MJD to ISOT."
        for mjd, isot in self.test_data:
            self.assertEqual(dateconvert.MJD_to_ISOT(mjd), isot)
```

unittest example (cont.)

```
    def test_ISOT_to_MJD(self):
        "Test conversion from ISOT to MJD."
        for mjd, isot in self.test_data:
            self.assertAlmostEqual(dateconvert.ISOT_to_MJD(isot), mjd, places=9

    def test_DateConversionConsistency(self):
        "Test the round trip between MJD_to_ISOT and ISOT_to_MJD."
        for mjd in np.arange(40000, 57700, 31.1424, dtype=np.float):
            isot = dateconvert.MJD_to_ISOT(mjd)
            self.assertAlmostEqual(mjd, dateconvert.ISOT_to_MJD(isot), places=9

    def test_ISOT_to_MJD_bad_input(self):
        "Pass the correct type (str), but with an invalid value."
        isot = '10-07-2016 00:00:00'
        self.assertRaises(ValueError, dateconvert.ISOT_to_MJD, isot)

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

unittest example (cont.)

Running this one gets

```
$ python test_dateconvert.py
F...
======================================================================
FAIL: test_DateConversionConsistency (__main__.DateConversionTestCase)
Test the round trip between MJD_to_ISOT and ISOT_to_MJD.
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test_dateconvert.py", line 29, in test_DateConversionConsistency
    self.assertAlmostEqual(mjd, dateconvert.ISOT_to_MJD(isot), places=9)
AssertionError: 45169.638399999501 != 45169.6384 within 9 places

----------------------------------------------------------------------
Ran 4 tests in 0.053s

FAILED (failures=1)
$
```

Continuous Integration

- Tests at all levels are run regularly over the whole code base to ensure that the software is in an operational state.
- CI server testing can include static code checking, coverage assessment, unit tests, integration tests, performance tests, and aggregation of metrics including time histories.
- CI systems such as Travis-CI and Jenkins can be run to trigger on code respository events, such as commits or pull-requests, or to run on regular intervals (nightly builds).
- In order for CI to be effective, work should be committed and tested on at least a daily basis in order to prevent conflicts which are difficult to resolve and which stem from large commits, e.g., a week's worth of work.
- For revision systems where branching is available, CI builds can be performed for the branch against the current master, and should not be merged unless it builds correctly.
- These practices help keep the mainline of the code in a deployable state at all times, and provide current builds for testing by clients.

Using Travis-CI

- Free for public GitHub-hosted repositories
  - Add a .travis.yml file.
    - install code and dependencies
    - set up environment
    - run tests and coverage analysis
  - Connect GitHub repo to Travis-CI.
    Go to your GitHub repo (as admin):
    Settings -> Integrations & services -> Add service
    At Travis-CI:
    My Repositories +(Add New Repository) -> Sync Account ->
    <Activate switch>
  - At GitHub: Settings -> Integrations & services -> Travis-CI -> Test service
  - Connect to Coveralls -> Add Repos -> Refresh Private Repos -> toggle repo switch
  - Add badges to GitHub repo.

Example .travis.yml

```
language: python

python:
  - "2.7"

install:
  - pip install -r requirements.txt
  - source setup/setup.sh

services:
  - mysql

before_script:
  - mysql -e 'create database myapp_test;'
  - mysql -e 'show databases;'

script:
  - nosetests -s --with-coverage --cover-package=dateconvert
  - pylint --py3k `find . -name \*.py -print`

after_success:
  - coveralls
```

Some resources on unit testing

- http://docs.python-guide.org/en/latest/writing/tests/
- https://docs.python.org/2/library/unittest.html
- http://www.diveintopython.net/unit_testing/index.html
- https://travis-ci.org/
- https://coveralls.io/