Unit Testing in Python

Robert J. Hansen

December 5, 2014

Unit testing is one of those things that we all say we think is a good idea, and yet we write few of them. In this L&L, we'll cover what unit testing is, what it isn't, and how to use Python's built-in `unittest` module.

Contents

Overview

Unit tests are useful because they tell us something very important: when are we finished writing code? When have we reached our objective? When can we hand this off to SQA for further work?

Further, unit tests help us by permitting us to see at a glance how much work remains to be done, and where. Want to see what parts of the code need love? Check the last run of the unit tests.

Finally, unit tests make it easy to avoid regressions (where a bug fixed long-ago materializes again in the code). When we find bugs in our code, we can write a unit test to check for the presence of this bug. If later changes to the code reintroduce the bug, the unit test will flag it and thus help us avoid reintroducing errors.

What should be tested?

Short answer: "if it can be tested in an automated fashion, write a unit test for it."

Yes, this will mean your unit tests will quickly dwarf your code — at least in terms of lines of code. That's neither a problem nor unexpected. The SQLite database system, for instance, has ten lines of unit tests for every line of code.

How should it be tested?

Short answer: “with either as little or as much logging as possible.” Generally speaking, a test that passes should produce no output, but a test that fails should produce a log containing enough information to help the programmer isolate the problem.

When should it be tested?

As soon as possible, preferably before a single line of code is written!

Example: libgenquad

Let’s assume Management has come down with a requirement: we must write a Python module that implements the General Quadratic Equation. For those who’ve forgotten high-school algebra, a quadratic equation is one that looks like $Ax^2 + Bx + C = 0$.[1] The General Quadratic Equation takes in three parameters of A, B, and C, and yields the values of x satisfy that formula. The General Quadratic Equation itself is pretty compact: $\frac{-b\pm\sqrt{b^2-4ac}}{2a}$.

[1] A, B, and C are allowed to be any number. Integers, reals, complex, the GQE handles them all. However, for sake of this discussion we’re going to assume we only need to support floating-point values.

We begin by asking, “what must our code do to be a success?” At the very least it must:

- Give correct answers on valid inputs
- Raise an exception if we pass in an A value of zero[2]
- Raise an exception if there are no real solutions[3]

[2] If $A = 0$ then we have a linear equation, not a quadratic; further, in the denominator of the GQE we have to divide by $2A$. Allowing $A = 0$ would lead to a division-by-zero error.

[3] E.g., there is no real solution to the quadratic equation $x^2 + 1 = 0$; there are complex solutions at $\pm\imath$, however.

So, without further ado, let’s write a code skeleton. We need two exception classes to handle the error cases of “complex solutions found” and “this is not a quadratic equation”; we need an empty stub function to handle the actual general quadratic equation; and then we need to create a test case.

Unit test 1

```python
#!/usr/bin/python3
#coding=UTF-8

from unittest import TestCase, main
from random import seed, uniform
from libgenquad import *

class TestGenQuad(TestCase):
    def test_not_quadratic(self):
        with self.assertRaises(NotQuadratic):
            solve_quadratic(0, 1, 1)

    def test_correct_answers(self):
        for run in range(0, 1000):
            M = uniform(-100, 100)
            N = uniform(-100, 100)
            distortion = uniform(-100, 100)
            A = 1 * distortion
            B = (-(M + N)) * distortion
            C = M * N * distortion

            solutions = solve_quadratic(A, B, C)

            self.assertTrue(type(solutions)
                            is tuple)
            self.assertTrue(type(solutions[0])
                            is float)
            self.assertTrue(type(solutions[1])
                            is float)

            M_in_sols = [X for X in solutions
                         if abs(X-M) < .00001]
            N_in_sols = [X for X in solutions
                         if abs(X-N) < .00001]

            self.assertTrue(M_in_sols)
            self.assertTrue(N_in_sols)

    def test_only_reals(self):
        A = 1
        B = 0
        C = 1
        with self.assertRaises(ComplexSolutions):
            solve_quadratic(A, B, C)

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

Already, we can see the test case is going to be much larger than the actual code is. Implementing the general quadratic equation will only be a few lines of code: testing it requires a lot more.

Within the `unittest` module there are two things that must be imported for any non-trivial unit test. The first is `unittest.TestCase`, and the second is `unittest.main`. The former is the parent class that all unit tests must inherit from, and the latter is a function that walks through every class that inherits from `unittest.TestCase` and runs every method found within them that starts with `test_`.

The `unittest.TestCase` class provides a lot of functionality to us. What most interests of are the methods `assertTrue`, `assertFalse`, `assertRaises`, and many others.[4] With the exception of `assertRaised`, they all work more or less identically: they evaluate whatever is passed to them, and raises an exception if whatever condition is being tested for doesn't come to pass.

[4] `assertEqual`, `assertNotEqual`, `assertIs`, `assertIsNot`, `assertIsNone`, `assertIsNotNone`, `assertIn`, `assertNotIn`, `assertIsInstance`, `assertNotIsInstance` round out the list of commonly used assertion methods.

`assertRaised` is slightly different. That gets placed inside a `with` block. If the specified exception hasn't occurred by the end of the `with` block, the unit test fails.

The `unittest` framework traps all of these exceptions and uses them to come up with a concise output of which tests succeeded, which tests failed, and why.

Now that we have the test framework written, let's finish the code by writing `libgenquad`.

Unit test 2

```python
#!/usr/bin/python3
#coding=UTF-8

from math import sqrt

class ComplexSolutions(Exception):
    def __init__(self):
        Exception.__init__(self)

class NotQuadratic(Exception):
    def __init__(self):
        Exception.__init__(self)

def solve_quadratic(a, b, c):
    if a == 0:
        raise NotQuadratic()
    discriminant = b**2 - 4*a*c
    if discriminant < 0:
        raise ComplexSolutions()
    numerator1 = -b + sqrt(discriminant)
    numerator2 = -b - sqrt(discriminant)
    denominator = 2*a
    return (numerator1 / denominator,
            numerator2 / denominator)
```

As can be seen here, we have about ten lines of code that implements the library but considerably more than that testing the library. Don't worry about this: once you have a little experience in writing test cases they're quick and easy to write and take very little time. They can save you orders of magnitude more time than it takes to write them.

Best practices

1. **If it can be automated, automate it.** Some things can't easily be automated; for instance, GUI interactions usually have to be done by hand.
2. **Write unit tests before you write code.** If you write the code first, the unit tests will never be completed. Code always gets finished right before deadlines, which means there's never enough time to write a proper unit testing framework.
3. **Connect unit tests to functional requirements.** If your program must do something, write a unit test to make sure it does it.
4. **Bugs get unit tests.** If a user reports a bug, find a way to test for the bug's presence and add that to the unit tests.
5. **Run them frequently.** At the very least run them twice a day: once when you begin working on the code, so that you get a good idea of what functionality is present and what needs work, and once when you finish, so you can get an idea of what you've achieved.
6. **Talk to SQA.** They have their own idea of what needs to be tested, and it's probably better than ours is. They're the experts in this area: lean on them.