Improve Your Python: Understanding Unit Testing

Jeff Knupp Dec 09, 2013

One frequent source of confusion for novice developers is the subject of testing. They are vaguely aware that "unit testing" is something that's good and that they should do, but don't understand what the term actually means. If that sounds like you, fear not! In this article, I'll describe what `unit testing` is, why it's useful, and how to `unit test` Python code.

What is Testing?

Before discussing *why* testing is useful and *how* to do it, let's take a minute to define *what* `unit testing` actually is. "Testing", in general programming terms, is the practice of writing code (separate from your actual application code) that invokes the code it tests to help determine if there are any errors. It **does not** prove that code is correct (which is only possible under very restricted circumstances). It merely reports if the conditions the tester thought of are handled correctly.

Note: when I use the term "testing", I'm always referring to "automated testing", where the tests are run by the machine. "Manual testing", where a human runs the program and interacts with it to find bugs, is a separate subject.

What kinds of things can be caught in testing? **Syntax errors** are unintentional misuses of the language, like the extra `.` in `my_list..append(foo)`. **Logical errors** are created when the algorithm (which can be thought of as "the way the problem is solved") is not correct.

Perhaps the programmer forgot that Python is "zero-indexed" and tried to print the last character in a string by writing `print(my_string[len(my_string)])` (which will cause an `IndexError` to be raised). Larger, more systemic errors can also be checked for. Perhaps the program always crashes when the user inputs a number greater than `100`, or hangs if the web site it's retrieving is not available.

All of these errors can be caught through careful testing of the code. `Unit testing`, specifically tests a single "unit" of code **in isolation**. A `unit` could be an entire module, a single class or function, or almost anything in between. What's important, however, is that the code is isolated from *other* code we're not testing (which itself could have errors and would thus confuse test results). Consider the following example:

```
def is_prime(number):
    """Return True if *number* is prime."""
    for element in range(number):
        if number % element == 0:
            return False

    return True

def print_next_prime(number):
    """Print the closest prime number larger than *number*."""
    index = number
    while True:
        index += 1
        if is_prime(index):
            print(index)
```

We have two functions, `is_prime` and `print_next_prime`. If we wanted to test `print_next_prime`, we would need to be sure that `is_prime` is correct, as `print_next_prime` makes use of it. In this case, the function `print_next_prime` is one unit, and `is_prime` is another. Since `unit tests`

test only a **single** unit at a time, we would need to think carefully about how we could accurately test `print_next_prime` (more on how this is accomplished later).

So what does test code look like? If the previous example is stored in a file named `primes.py`, we may write test code in a file named `test_primes.py`. Here are the minimal contents of `test_primes.py`, with an example test:

```
import unittest
from primes import is_prime

class PrimesTestCase(unittest.TestCase):
    """Tests for `primes.py`."""

    def test_is_five_prime(self):
        """Is five successfully determined to be prime?"""
        self.assertTrue(is_prime(5))

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

The file creates a `unit test` with a single `test case`: `test_is_five_prime`. Using Python's built-in `unittest` framework, any member function whose name begins with `test` in a class deriving from `unittest.TestCase` will be run, and its assertions checked, when `unittest.main()` is called. If we "run the tests" by running `python test_primes.py`, we'll see the output of the `unittest` framework printed on the console:

```
$ python test_primes.py
E
======================================================================
ERROR: test_is_five_prime (__main__.PrimesTestCase)
----------------------------------------------------------------------
Traceback (most recent call last):
File "test_primes.py", line 8, in test_is_five_prime
```

```
    self.assertTrue(is_prime(5))
File "/home/jknupp/code/github_code/blug_private/primes.py", line 4, in is_pri
    if number % element == 0:
ZeroDivisionError: integer division or modulo by zero

----------------------------------------------------------------------
Ran 1 test in 0.000s
```

The single "E" represents the results of our single test (if it was successful, a "." would have been printed). We can see that our test failed, the line that caused the failure, and any exceptions raised.

Why Testing?

Before we continue with the example, it's important to ask the question, "Why is testing a valuable use of my time?" It's a fair question, and it's the question those unfamiliar with testing code often ask. After all, testing takes time that could otherwise be spend writing code, and isn't that the most productive thing to be doing?

There are a number of valid answers to this question. I'll list a few here:

Testing makes sure your code works properly under a given set of conditions

Testing assures correctness under a basic set of conditions. Syntax errors will almost certainly be caught by running tests, and the basic logic of a unit of code can be tested to ensure correctness under certain conditions. Again, it's not about proving the code is correct *under any set of conditions*. We're simply aiming for a reasonably complete set of possible conditions (i.e. you may write a test for what happens when you call `my_addition_function(3, 'refrigerator`), but you needn't test all possible strings for each argument).

Testing allows one to ensure that changes to the code did not break

existing functionality

This is especially helpful when `refactoring`[1] code. Without tests in place, you have no assurances that your code changes did not break things that were previously working fine. **If you want to be able to change or rewrite your code and know you didn't break anything, proper unit testing is imperative.**

Testing forces one to think about the code under unusual conditions, possibly revealing logical errors

Writing tests forces you to think about the non-normal conditions your code may encounter. In the example above, `my_addition_function` adds two numbers. A simple test of basic correctness would call `my_addition_function(2, 2)` and assert that the result was `4`. Further tests, however, might test that the function works correctly with `float`s by running `my_addition_function(2.0, 2.0)`. *Defensive coding* principles suggest that your code should be able to gracefully fail on invalid input, so testing that an exception is properly raised when strings are passed as arguments to the function.

Good testing requires modular, decoupled code, which is a hallmark of good system design

The whole practice of unit testing is made much easier by code that is *loosely coupled*[2]. If your application code has direct database calls, for example, testing the logic of your application depends on having a valid database connection available and test data to be present in the database. Code that isolates external resources, on the other hand, can easily replace them during testing using *mock objects*. Applications designed with test-ability in mind usually end up being modular and loosely coupled out of

necessity.

The Anatomy of A Unit Test

We'll see how to write and organize unit tests by continuing the example from earlier. Recall that `primes.py` contains the following code:

```
def is_prime(number):
    """Return True if *number* is prime."""
    for element in range(number):
        if number % element == 0:
            return False

    return True

def print_next_prime(number):
    """Print the closest prime number larger than *number*."""
    index = number
    while True:
        index += 1
        if is_prime(index):
            print(index)
```

The file `test_primes.py`, meanwhile, contains the following:

```
import unittest
from primes import is_prime

class PrimesTestCase(unittest.TestCase):
    """Tests for `primes.py`."""

    def test_is_five_prime(self):
        """Is five successfully determined to be prime?"""
        self.assertTrue(is_prime(5))

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

Making Assertions

`unittest` is part of the Python standard library and a good place to start our `unit test` walk-through. A `unit test` consists of one or more *assertions* (statements that assert that some property of the code being tested is true). Recall from your grade school days that the word "assert" literally means, "to state as fact." This is what assertions in `unit tests` do as well.

`self.assertTrue` is rather self explanatory, it asserts that the argument passed to it evaluates to `True`. The `unittest.TestCase` class contains a number of assert methods, so be sure to check the list and pick the appropriate methods for your tests. Using `assertTrue` for every test should be considered an anti-pattern as it increases the cognitive burden on the reader of tests. Proper use of `assert` methods state explicitly exactly what is being asserted by the test (e.g. it is clear what `assertIsInstance` is saying about its argument just by glancing at the method name).

Each test should test a single, specific property of the code and be named accordingly. To be found by the `unittest` test discovery mechanism (present in Python 2.7+ and 3.2+), test methods should be prepended by `test_` (this is configurable, but the purpose is to differentiate test methods and non-test utility methods). If we had named the method `test_is_five_prime` `is_five_prime` instead, the output when running `python test_primes.py` would be the following:

```
$ python test_primes.py

----------------------------------------------------------------------
Ran 0 tests in 0.000s

OK
```

Don't be fooled by the "OK" in the output above. "OK" is only reported because no tests actually ran! I think running zero tests should result in an error, but personal feelings aside, this is behavior you should be aware of, especially when programatically running and inspecting test results (e.g. with a *continuous integration* tool like TravisCI).

Exceptions

Returning to the *actual* contents of `test_primes.py`, recall that the output of `python test_primes.py` is the following:

```
$ python test_primes.py
E
======================================================================
ERROR: test_is_five_prime (__main__.PrimesTestCase)
----------------------------------------------------------------------
Traceback (most recent call last):
File "test_primes.py", line 8, in test_is_five_prime
    self.assertTrue(is_prime(5))
File "/home/jknupp/code/github_code/blug_private/primes.py", line 4, in is_pri
    if number % element == 0:
ZeroDivisionError: integer division or modulo by zero

----------------------------------------------------------------------
Ran 1 test in 0.000s
```

This output shows us that our single test resulted in failure due **not** to an assertion failing but rather because an un-caught exception was raised. In fact, the `unittest` framework didn't get a chance to properly run our test because it raised an exception before returning.

The issue here is clear: we are preforming the `modulo` operation over a range of numbers that includes zero, which results in a division by zero being performed. To fix this, we simply change the range to begin at `2` rather than

`0`, noting that modulo by `0` would be an error and modulo by `1` will always be True (and a prime number is one wholly divisible only by itself *and* 1, so we needn't check for `1`).

Fixing Things

A failing test has resulted in a code change. Once we fix the error (changing the line in `is_prime` to `for element in range(2, number):`), we get the following output:

```
$ python test_primes.py
.
----------------------------------------------------------------------
Ran 1 test in 0.000s
```

Now that the error is fixed, does that mean we should delete the test method `test_is_five_prime` (since clearly it will now always pass)? **No**. `unit tests` should rarely be deleted as *passing* tests are the end goal. We've tested that the syntax of `is_prime` is valid and, at least in one case, it returns the proper result. Our goal is to build a *suite* (a logical grouping of `unit tests`) of tests that all pass, though some may fail at first.

`test_is_five_prime` worked for an "un-special" prime number. Let's make sure it works for non-primes as well. Add the following method to the `PrimesTestCase` class:

```
def test_is_four_non_prime(self):
    """Is four correctly determined not to be prime?"""
    self.assertFalse(is_prime(4), msg='Four is not prime!')
```

Note that this time we added the optional `msg` argument to the `assert` call. If this test had failed, our message would have been printed to the console,

giving additional information to whoever ran the test.

Edge Cases

We've successfully tested two general cases. Let us now consider *edge cases*, or cases with unusual or unexpected input. When testing a function that whose range is all positive integers, examples of edge cases include `0`, `1`, a negative number, and a very large number. Let's test some of these now.

Adding a test for zero is straightforward. We expect `is_prime(0)` to return `False`, since, by definition, prime numbers must be greater than one:

```
def test_is_zero_not_prime(self):
    """Is zero correctly determined not to be prime?"""
    self.assertFalse(is_prime(0))
```

Alas, the output is:

```
python test_primes.py
..F
======================================================================
FAIL: test_is_zero_not_prime (__main__.PrimesTestCase)
Is zero correctly determined not to be prime?
----------------------------------------------------------------------
Traceback (most recent call last):
File "test_primes.py", line 17, in test_is_zero_not_prime
    self.assertFalse(is_prime(0))
AssertionError: True is not false

----------------------------------------------------------------------
Ran 3 tests in 0.000s

FAILED (failures=1)
```

Zero is incorrectly determined to be prime . We forgot that we decided to

skip checks of zero and one in our `range`. Let's add a special check for them:

```
def is_prime(number):
    """Return True if *number* is prime."""
    if number in (0, 1):
        return False

    for element in range(2, number):
        if number % element == 0:
            return False

    return True
```

The tests now pass. How will our function handle a negative number? It's important to know *before writing the test* what the output *should* be. In this case, any negative number should return `False`:

```
def test_negative_number(self):
    """Is a negative number correctly determined not to be prime?"""
    for index in range(-1, -10, -1):
        self.assertFalse(is_prime(index))
```

Here we decide to check all numbers from `-1 .. -9`. Calling a `test` method in a loop is perfectly valid, as are multiple calls to assert methods in a single test. We could have rewritten this in the following (more verbose) fashion:

```
def test_negative_number(self):
    """Is a negative number correctly determined not to be prime?"""
    self.assertFalse(is_prime(-1))
    self.assertFalse(is_prime(-2))
    self.assertFalse(is_prime(-3))
    self.assertFalse(is_prime(-4))
    self.assertFalse(is_prime(-5))
    self.assertFalse(is_prime(-6))
    self.assertFalse(is_prime(-7))
    self.assertFalse(is_prime(-8))
```

```
    self.assertFalse(is_prime(-9))
```

The two are essentially equivalent. Except when we run the loop version, we get a little less information than we'd like:

```
python test_primes.py
...F
======================================================================
FAIL: test_negative_number (__main__.PrimesTestCase)
Is a negative number correctly determined not to be prime?
----------------------------------------------------------------------
Traceback (most recent call last):
File "test_primes.py", line 22, in test_negative_number
    self.assertFalse(is_prime(index))
AssertionError: True is not false

----------------------------------------------------------------------
Ran 4 tests in 0.000s

FAILED (failures=1)
```

Hmm, we know the test failed, *but on which negative number*? Rather unhelpfully, the Python `unittest` framework does not print out the *expected* and *actual* values. We can side-step the issue in one of two ways: through the `msg` parameter or through the use of a third-party unit testing framework.

Using the `msg` parameter to `assertFalse` is simply a matter of recognizing that we can use string formatting to solve our problem:

```
def test_negative_number(self):
    """Is a negative number correctly determined not to be prime?"""
    for index in range(-1, -10, -1):
        self.assertFalse(is_prime(index), msg='{} should not be determined to
```

which gives the following output:

```
python test_primes
...F
======================================================================
FAIL: test_negative_number (test_primes.PrimesTestCase)
Is a negative number correctly determined not to be prime?
----------------------------------------------------------------------
Traceback (most recent call last):
File "./test_primes.py", line 22, in test_negative_number
    self.assertFalse(is_prime(index), msg='{} should not be determined to be p
AssertionError: True is not false : -1 should not be determined to be prime

----------------------------------------------------------------------
Ran 4 tests in 0.000s

FAILED (failures=1)
```

Fixing Code *Properly*

We see that the failing negative number was the first tested: `-1`. To fix this, we could add yet another special check for negative numbers, but the purpose of writing `unit tests` is not to blindly add code to check for edge cases. When a test fails, take a step back and determine the *best* way to fix the issue. In this case, rather than adding an additional `if`:

```
def is_prime(number):
    """Return True if *number* is prime."""
    if number < 0:
        return False

    if number in (0, 1):
        return False

    for element in range(2, number):
        if number % element == 0:
            return False

    return True
```

the following should be preferred:

```
def is_prime(number):
    """Return True if *number* is prime."""
    if number <= 1:
        return False

    for element in range(2, number):
        if number % element == 0:
            return False

    return True
```

In the latter, we recognize that the two `if` statements can be merged into a single statement that returns `False` if the argument is less than 1. This is both more succinct and properly aligned with the definition of a prime number (a number *greater than one* wholly divisible only by one and itself).

Third-Party Test Frameworks

We could have also solved the problem of too little information on test failure by using a third-party testing framework. The two most used are py.test and nose. Running our tests with `py.test -l` (`-l` "shows the values of local variables") gives the following:

```
#! bash

py.test -l test_primes.py
============================= test session starts ==============================
platform linux2 -- Python 2.7.6 -- pytest-2.4.2
collected 4 items

test_primes.py ...F

=================================== FAILURES ===================================
_____________________ PrimesTestCase.test_negative_number ______________________
```

```
self = <test_primes.PrimesTestCase testMethod=test_negative_number>

    def test_negative_number(self):
        """Is a negative number correctly determined not to be prime?"""
        for index in range(-1, -10, -1):
>           self.assertFalse(is_prime(index))
E           AssertionError: True is not false

index      = -1
self       = <test_primes.PrimesTestCase testMethod=test_negative_number>

test_primes.py:22: AssertionError
```

A bit more useful, as you can see. These frameworks provide far more functionality than simply more verbose output, but the point is just to be aware that they exist and extend the functionality of the built-in `unittest` package.

Wrapping Up

In this article, you learned *what* `unit tests` are, *why* they're important, and *how* to write them. That said, be aware we've merely scratched the surface of the topic of testing methodologies. More advanced topics such as *test case organization, continuous integration,* and *test case management* are good subjects for readers interested in further studying testing in Python.

1. Reorganizing/cleaning up code without changing functionality ↩
2. Code that does not expose its internal data or functions and does not make use of the internal data or functions of other code ↩

Discuss Posts With Other Readers at discourse.jeffknupp.com!

« My Development Environment For Python