Part 6: Developing the Test Further

Bookmark this page

Part 6: Developing the Test Further

The TestCase class includes instance methods for making assertions about the behavior of our program. The *assertEqual* method is an instance method defined in the TestCase class: it helps us compare the expected behavior of a program with its actual behavior. The first argument we give it is the *expected* result of squaring our number *num*. Then we give it the *actual* square that's produced by Squarer.calc. We can also specify an optional helpful message for the test library to print if this test fails: our helpful message will tell us what number we were trying to square when the test failed.

Here's another difference: note that I didn't include an *if __name__=="__main__"* clause. We *won't* be running this script directly, instead we'll invoke it indirectly through the unittest library. You'll see that below.

Finally, there's one more difference between unittest and our original test script that you should be aware of: unittest will stop running a test method as soon as it finds a single assertion error. That is to say, if test_positive_numbers finds that Squarer does not produce 9 when squaring 3, then it *will* report that error but then it won't try any more numbers: it **won't** also try squaring 12 to see if it produces 144. The implication is that, for code that is more complex it is possible that fixing one error reported by unittest will uncover a new one.

Here's the current content of our squarer.py file:

```
# squarer.py
class Squarer(object):

    @staticmethod
    def calc(operand):
        return operand**2
```

I expect that our Squarer.calc method should be working correctly! And here's the output of running our test:

```
$ python -m unittest test2
..
----------------------------------------------------------------------
Ran 2 tests in 0.000s

OK
$
```

Excellent! Our Squarer.calc method is working as expected. We wrote two tests, test_positive_numbers and test_negative_numbers, and our unittest script found no discrepancies between the expected and actual behavior of Squarer.calc. Each test method that our code satisfies is represented by a '.' at the top of the output. The "OK" also signifies that each test was passed by our code. Finally, if you're a UNIX geek, you'll be interested to know that this test run produced a zero return value.

We can also choose which tests to run. Calling *python -m unittest test2* runs all of the tests in the test2.py file. If there were multiple test classes in the test2 file, we could choose to run only the SquarerTest tests:

```
$ python -m unittest test2.SquarerTest
..
----------------------------------------------------------------------
Ran 2 tests in 0.001s

OK
```

We can also choose to run just a single test within the SquarerTest class:

```
$ python -m unittest test2.SquarerTest.test_positive_numbers
.
----------------------------------------------------------------------
```

```
Ran 1 test in 0.000s

OK
```

Running just a single test at a time can be helpful if all of your tests together take a long time to run, but you're working on fixing a single bug that's covered by a single test. It is also useful if multiple tests are failing and you are only working on one test.

Let's modify our squarer.py file to produce an error and see how unittest reports test failures. Modify squarer.py:

```
# squarer.py
class Squarer(object):

    @staticmethod
    def calc(operand):
        return operand**2  # OLD
        return operand**operand
```

Running our tests produces:

```
$ python -m unittest test2

FF
######################################################################
FAIL: test_negative_numbers (test2.SquarerTest)
----------------------------------------------------------------------
Traceback (most recent call last):
 File "/Users/jaschilz/tmp/test2.py", line 32, in test_negative_numbers
 self.assertEqual(square, Squarer.calc(num), "Squaring {}".format(num));
AssertionError: 1 !# -1.0 : Squaring -1

######################################################################
FAIL: test_positive_numbers (test2.SquarerTest)
----------------------------------------------------------------------
Traceback (most recent call last):
 File "/Users/jaschilz/tmp/test2.py", line 19, in test_positive_numbers
 self.assertEqual(square, Squarer.calc(num), "Squaring {}".format(num));
AssertionError: 9 !# 27 : Squaring 3


----------------------------------------------------------------------
Ran 2 tests in 0.001s

FAILED (failures#2)
```

Each test that failed is represented by an "F" at the top of the output. If we are running multiple tests, with some passes and some failures, then we would see a mix of "."s and "F"s at the top of the output. In this case, we ran two tests and both failed. If our code fails *any* tests, then we will also see the word "FAILED" at the bottom of the output, replacing "OK". If you're a UNIX geek, you might be interested to know that this test run has produced a non-zero return value.

The unittest library also gives us detailed information about each test that failed. Let's look at the output for test_positive_numbers:

```
######################################################################
FAIL: test_positive_numbers (test2.SquarerTest)
----------------------------------------------------------------------
Traceback (most recent call last):
 File "/Users/jaschilz/tmp/test2.py", line 19, in test_positive_numbers
 self.assertEqual(square, Squarer.calc(num), "Squaring {}".format(num));
AssertionError: 9 !# 27 : Squaring 3


----------------------------------------------------------------------
```

We can see that the code failed its assertion on line 19. The unittest library reports that the expected value, 9, was not equal to the actual value of 27 produced by our code. We also see the helpful output message that we created: "Squaring

3”. This tells us that the test failed while attempting our test scenario for squaring the number 3.

Keep in mind that unittest will stop a test method as soon as it encounters its first assertion error! Our Squarer.calc would probably *also* fail to produce 144 when squaring 12, but our test method will not move on to that scenario until our code passes the scenario for squaring 3.

Now that we know that our change to squarer.py has introduced an error, let’s revise our code to fix the error, re-run the tests, and see that our code is working as expected once again.

```
# squarer.py
class Squarer(object):

    @staticmethod
    def calc(operand):
        #return operand**2        # OLD
        #return operand**operand  # BAD
        return operand*operand   # This should work
```

Running our tests:

```
$ python -m unittest test2
..
----------------------------------------------------------------------
Ran 2 tests in 0.001s

OK
```

Great! Our squarer works as expected again!

In practice you’ll probably always use unittest or another similar library instead of your own, completely homegrown test