Test Driven Development with unittest

WE'LL COVER THE FOLLOWING

In this section, you will learn about Test Driven Development (TDD) with Python using Python's built-in unittest module. I do want to thank Matt and Aaron for their assistance in showing me how TDD works in the real world. To demonstrate the concepts of TDD, we'll be covering how to score bowling in Python. You may want to use Google to look up the rules for bowling if you don't already know them. Once you know the rules, it's time to write some tests. In case you didn't know, the idea behind Test Driven Development is that you write the tests BEFORE you write the actual code. In this chapter, we will write a test, then some code to pass the test. We will iterate back and forth between writing tests and code until we're done. For this chapter, we'll write just three tests. Let's get started!

The First Test

Our first test will be to test our game object and see if it can calculate the correct total if we roll eleven times and only knock over one pin each time. This should give us a total of eleven.

```
import unittest

class TestBowling(unittest.TestCase):
    """"""

    def test_all_ones(self):
        """Constructor"""
        game = Game()
        game.roll(11, 1)
```

```
        self.assertEqual(game.score, 11)
```

This is a pretty simple test. We create a game object and then call its **roll** method eleven times with a score of one each time. Then we use the **assertEqual** method from the **unittest** module to test if the game object's score is correct (i.e. eleven). The next step is to write the simplest code you can think of to make the test pass. Here's one example:

```
class Game:
    """

    def __init__(self):
        """Constructor"""
        self.score = 0

    def roll(self, numOfRolls, pins):
        """
        for roll in numOfRolls:
            self.score += pins
```

For simplicity's sake, you can just copy and paste that into the same file with your test. We'll break them into two files for our next test. Anyway, as you can see, our **Game** class is super simple. All that was needed to pass the test was a score property and a **roll** method that can update it.

Let's run the test and see if it passes! The easiest way to run the tests is to add the following two lines of code to the bottom of the file:

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

Then just run the Python file via the command line, If you do, you should get something like the following:

```
E
----------------------------------------------------------------------
ERROR: test_all_ones (__main__.TestBowling)
Constructor
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\Users\Mike\Documents\Scripts\Testing\bowling\test_one.py",
 line 27, in test_all_ones
    game.roll(11, 1)
  File "C:\Users\Mike\Documents\Scripts\Testing\bowling\test_one.py",
 line 15, in roll
    for roll in numOfRolls:
TypeError: 'int' object is not iterable
```

```
TypeError: 'int' object is not iterable

----------------------------------------------------------------------

Ran 1 test in 0.001s

FAILED (errors=1)
```

Oops! We've got a mistake in there somewhere. It looks like we're passing an Integer and then trying to iterate over it. That doesn't work! We need to change our Game object's roll method to the following to make it work:

```
def roll(self, numOfRolls, pins):
    """"""
    for roll in range(numOfRolls):
        self.score += pins
```

If you run the test now, you should get the following:

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

OK
```

Note the "." because it's important. That little dot means that one test has run and that it passed. The "OK" at the end clues you into that fact as well. If you study the original output, you'll notice it leads off with an "E" for error and there's no dot! Let's move on to test #2.

The Second Test

For the second test, we'll test what happens when we get a strike. We'll need to change the first test to use a list for the number of pins knocked down in each frame though, so we'll look at both tests here. You'll probably find this to be a fairly common process where you may need to edit a couple of tests due to fundamental changes in what you're testing for. Normally this will only happen at the beginning of your coding and you will get better later on such that you shouldn't need to do this. Since this is my first time doing this, I wasn't thinking far enough ahead. Anyway, let's take a look at the code:

```
from game import Game
import unittest

class TestBowling(unittest.TestCase):
```

```
class TestBowling(unittest.TestCase):
    """



    def test_all_ones(self):
        """Constructor"""
        game = Game()
        pins = [1 for i in range(11)]
        game.roll(11, pins)
        self.assertEqual(game.score, 11)

    def test_strike(self):
        """
        A strike is 10 + the value of the next two rolls. So in this case
        the first frame will be 10+5+4 or 19 and the second will be
        5+4. The total score would be 19+9 or 28.
        """
        game = Game()
        game.roll(11, [10, 5, 4])
        self.assertEqual(game.score, 28)

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

Let's take a look at our first test and how it changed. Yes, we're breaking the rules here a bit when it comes to TDD. Feel free to NOT change the first test and see what breaks. In the **test_all_ones** method, we set the **pins** variable to equal a list comprehension which created a list of eleven ones. Then we passed that to our **game** object's **roll** method along with the number of rolls.

In the second test, we roll a strike in our first roll, a five in our second and a four in our third. Note that we went a head and told it that we were passing in eleven rolls and yet we only pass in three. This means that we need to set the other eight rolls to zeros. Next, we use our trusty **assertEqual** method to check if we get the right total. Finally, note that we're now importing the **Game** class rather than keeping it with the tests. Now we need to implement the code necessary to pass these two tests. Let's take a look at one possible solution:

```
class Game:
    """



    def __init__(self):
        """Constructor"""
        self.score = 0
        self.pins = [0 for i in range(11)]

    def roll(self, numOfRolls, pins):
        """
        x = 0
        for pin in pins:
            self.pins[x] = pin
```

```
            x += 1
        x = 0
        for roll in range(numOfRolls):
            if self.pins[x] == 10:
                self.score = self.pins[x] + self.pins[x+1] + self.pins[x+2]
            else:
                self.score += self.pins[x]
            x += 1
        print(self.score)
```

Right off the bat, you will notice that we have a new class attribute called **self.pins** that holds the default pin list, which is eleven zeroes. Then in our **roll** method, we add the correct scores to the correct position in the self.pins list in the first loop. Then in the second loop, we check to see if the pins knocked down equals ten. If it does, we add it and the next two scores to score. Otherwise, we do what we did before. At the end of the method, we print out the score to check if it's what we expect. At this point, we're ready to code up our final test.

The Third (and Final) Test

Our last test will test for the correct score that would result should someone roll a spare. The test is easy, the solution is slightly more difficult. While we're at it, we're going to refactor the test code a bit. As usual, we will look at the test first.

```
from game_v2 import Game
import unittest

class TestBowling(unittest.TestCase):
    """"""

    def setUp(self):
        """"""
        self.game = Game()

    def test_all_ones(self):
        """
        If you don't get a strike or a spare, then you just add up the
        face value of the frame. In this case, each frame is worth
        one point, so the total is eleven.
        """
        pins = [1 for i in range(11)]
        self.game.roll(11, pins)
        self.assertEqual(self.game.score, 11)

    def test_spare(self):
        """
        A spare is worth 10, plus the value of your next roll. So in this
        case, the first frame will be 5+5+5 or 15 and the second will be
        5+4 or 9. The total is 15+9, which equals 24
```

```
        """
        self.game.roll(11, [5, 5, 5, 4])
        self.assertEqual(self.game.score, 24)

    def test_strike(self):
        """
        A strike is 10 + the value of the next two rolls. So in this case
        the first frame will be 10+5+4 or 19 and the second will be
        5+4. The total score would be 19+9 or 28.
        """
        self.game.roll(11, [10, 5, 4])
        self.assertEqual(self.game.score, 28)

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

First off, we added a **setUp** method that will create a self.game object for us for each test. If we were accessing a database or something like that, we would probably have a tear down method too for closing connections or files or that sort of thing. These are run at the beginning and end of each test respectively should they exist. The **test_all_ones** and **test_strike** tests are basically the same except that they are using “self.game” now. The only the new test is **test_spare**. The docstring explains how spares work and the code is just two lines. Yes, you can figure this out. Let’s look at the code we’ll need to pass these tests:

```
# game_v2.py

class Game:
    """"""

    def __init__(self):
        """Constructor"""
        self.score = 0
        self.pins = [0 for i in range(11)]

    def roll(self, numOfRolls, pins):
        """"""
        x = 0
        for pin in pins:
            self.pins[x] = pin
            x += 1
        x = 0
        spare_begin = 0
        spare_end = 2
        for roll in range(numOfRolls):
            spare = sum(self.pins[spare_begin:spare_end])
            if self.pins[x] == 10:
                self.score = self.pins[x] + self.pins[x+1] + self.pins[x+2]
            elif spare == 10:
                self.score = spare + self.pins[x+2]
                x += 1
            else:
                self.score += self.pins[x]
```

```
                self.score += self.pins[x]
            x += 1
            if x == 11:
                break
            spare_begin += 2
            spare_end += 2
        print(self.score)
```

For this part of the puzzle, we add to our conditional statement in our loop. To calculate the spare's value, we use the **spare_begin** and **spare_end** list positions to get the right values from our list and then we sum them up. That's what the **spare** variable is for. That may be better placed in the elif, but I'll leave that for the reader to experiment with. Technically, that's just the first half of the spare score. The second half are the next two rolls, which is what you'll find in the calculation in the elif portion of the current code. The rest of the code is the same.