Module 8 Additional Challenge: Unit Testing

JHU EP 606.206 - Introduction to Programming Using Python

Introduction

We saw in the Module 8 Live Demo Office Hour how a developer (Joe) can make changes to their class implementation without affecting the client code that makes use of the class (Alan); this idea is known as **abstraction**. However, while this provides many benefits, what happens if I make an update to my module that inadvertently breaks the code!? Generally speaking, it's a bad idea to make any changes to code, even simple changes, without testing them. But, while testing is always a good idea, it can be particularly onerous to test the code manually, especially if it is a large body of code. Wouldn't it be nice if we could write a set of tests and run them against our code automatically? Good news: Python provides a built-in library called 'unittest' that allows us to do exactly this. While it requires some effort up-front, once the test cases are written they can easily be executed to test any code changes going forward.

Python's *unittest*

Python's built-in unittest module allows you to write a suite of test cases and execute them against your code automatically. To do this you can create a new class that is a subclass of the unittest.TestCase class and defines how to setup and execute your test cases. Please read Real Python's article on Python Testing and refer to Python's official documentation on unittest to learn more about how to use unittest.

As a concrete example, let's think about unit testing in the context of the Module 8 LFSR Assignment. Before we can write our ***image_encrypter*** class, we need to make sure the ***LFSR*** class is functional. There are two primary class methods we need to test to ensure the LFSR is working properly: **`step()`** and **`bit()`**. A natural way to do this is to create two separate tests (classes), one for each method:

1. ***class LFSRStepTest*** that contains a single method:
    a. ***`test_step()`*** that takes the 5 LFSR's from the Assignment and tests them against the 5 resulting LFSR's after 1 call to the step() method (also provided in the Assignment)
        i. You'll find it useful to override the __eq__ method such that it returns `True` if two LFSR's have the same seed
2. ***class LFSRBitTest*** that contains one or both of the following methods:
    a. **`test_bit()`**: compare the values returned by **`bit()`** called on a static list of indices to their actual values within the seed
    b. **`test_bit_random()`**: compare the values returned by **`bit()`** called on a random list of indices to their actual values within the seed

NOTE: depending on how you write your test cases, they might require slight tweaks to your LFSR class. For example, I modified my LFSR's step() method to not only update the seed, but to also return the updated LFSR containing the updated seed value as well.

Here is a link to our solution and our results and some starter code is provided in the image below:

Starter Code

```
import math
import random
import unittest
from lfsr import LFSR

# convenience class that allows both test classes below to reuse a single set of test data
class LFSRTest():
    test_lfsrs = []  # LFSR's to test
    result_lfsrs = []  # known LFSR results after 1 step
    test_bits = [0, 3, 7]  # indexes of bits to test (left-right)
    result_bits = [[0, 0, 1], [0, 0, 0], [1, 1, 1], [0, 1, 1], [1, 0, 1]]  # LFSR bits from index 0, 3, 7

    # build test LFSR's
    test_lfsrs.append(LFSR(seed='0110100111', tap=2))
    test_lfsrs.append(LFSR(seed='0100110010', tap=8))
    test_lfsrs.append(LFSR(seed='1001011101', tap=5))
    test_lfsrs.append(LFSR(seed='0001001100', tap=1))
    test_lfsrs.append(LFSR(seed='1010011101', tap=7))

    # build results LFSR's (known LFSR state after 1 step)
    result_lfsrs.append(LFSR(seed='1101001111', tap=2))
    result_lfsrs.append(LFSR(seed='1001100100', tap=8))
    result_lfsrs.append(LFSR(seed='0010111010', tap=5))
    result_lfsrs.append(LFSR(seed='0010011000', tap=1))
    result_lfsrs.append(LFSR(seed='0100111011', tap=7))

################################################################################

# HINT: because LFSRStepTest is a subclass of LFSRTest you can access test_lfsrs and result_lfsrs in the superclass using the super() keyword
class LFSRStepTest(unittest.TestCase, LFSRTest):

    def test_step(self):
        # YOUR CODE HERE

################################################################################

# HINT: because LFSRBitTest is a subclass of LFSRTest you can access test_lfsrs and result_lfsrs in the superclass using the super() keyword
class LFSRBitTest(unittest.TestCase, LFSRTest):

    def test_bit(self):
        # YOUR CODE HERE

    def test_bit_random(self):
        # YOUR CODE HERE

################################################################################

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

Results