Mod 2 Lab: An autograder of your very own

By now, you've encountered the limitations of our autograder - it prints the results of our test cases, but those results can be difficult to interpret. In this assignment, you will create your own suite of test cases to serve as a local autograder. This is the core loop in this course: write a unittest, implement the funcitonality, repeat.

Test-Driven Development

Test-driven development, or TDD, involves three stages often referred to as Red-Green-Refactor:

- Red - write a test, then run your code to **verify that the test case fails**
- Green - modify your code until the test passes
- Refactor - extract duplicate algorithms/classes into a parent function/superclass.

Phase 1 - Red

For instance, let's say we are creating a class called `Point` to store a 2-D point. We might create the following skeleton code and unittest for our first TDD loop:

Point.py

```
class Point:
    def __init__(self, x, y):
        """Initializes a 2-D point with x- and y- coordinates"""
        pass
```

TestPoint.py

```
import unittest
from Point import Point # import class Point from file Point.py

class TestPoint(unittest.TestCase):
    def setUp(self):
        """Create some points for future tests"""
        self.p1 = Point(3, 4)
        self.p2 = Point(5, 6)

    def test_init(self):
        """Tests that points are initialied with the correct attributes"""
        self.assertEqual(self.p1.x, 3)
        self.assertEqual(self.p1.y, 4)

    def test_eq(self):
        """ADD YOUR OWN DOCSTRING"""

    def test_equidistant(self):
        """ADD YOUR OWN DOCSTRING"""

    def test_within(self):
        """ADD YOUR OWN DOCSTRING"""
```

Run `TestPoint.py`, and you should see something like the following:

```
$ python3 ./TestPoint.py
..E.
======================================================================
ERROR: test_init (__main__.TestPoint.test_init)
Tests that points are initialied with the correct attributes
----------------------------------------------------------------------
Traceback (most recent call last):
  File ".\TestPoint.py", line 12, in test_init
    self.assertEqual(self.p1.x, 3)
                     ^^^^^^^^
AttributeError: 'Point' object has no attribute 'x'
----------------------------------------------------------------------
Ran 4 tests in 0.001s

FAILED (errors=1)
```

Now, we celebrate! Our test fails - **it is a good test**, and we are ready to move on to phase 2 - Green.

Phase 2 - Green

Next, we implement the `init` method. Modify `Point.__init__` to store x- and y- attributes:

```
class Point:
    def __init__(self, x, y):
        """Initializes a 2-D point with x- and y- coordinates"""
        self.x = x
        self.y = y
```

Run our code again to make sure it passes:

```
python .\TestPoint.py
....
----------------------------------------------------------------------
Ran 4 tests in 0.000s

OK
```

Everything works! We don't have any code to refactor, so we are done with the first TDD loop. Now we write the second unittest, run it, implement functionality, and run it again.

Practice TDD *in pairs*

Your turn. Implement functionality one unittest at a time. Follow this flow:

1) Write a single unittest in `TestPoint.py`
2) Run `TestPoint.py` to verify you have 1 failure
3) Implement functionality in `Point.py`
4) Run `TestPoint.py` to verify you have 0 failures

You should use **pair programming** for this lab (during lab time - you'll have to do it solo if you are not

able to finish during lab). Only one partner should be logged on to their computer. The other partner looks on. The logged in partner (the *coder*) writes and runs the code, explaining what they are doing to the other partner (the *observer*), who can make suggestions and ask questions. Switch off after every test.

Partners should share their code using e.g. a cloud link or email at the end of class, since both partners need to submit individually to receive creit.

Use this flow until you have implemented the following functionality in `Point.py`:

- `Point.__eq__(self, other)`
  - returns `True` iff self and other have the same x and y attributes
  - Use at least **two** assertions in your unittest method `TestPoint.test_eq()`:
    - Use `self.assertEqual()` to confirm that 2 points with the same x- and y- compare as equal
    - Use `self.assertNotEqual()` to confirm that 2 points with different x- and y- do not compare as equal
- `Point.equidistant(self, other)`:
  - returns `True` iff self and other are the same distance from the origin
    - Use $sqrt(x^2 + y^2)$ to find the distance from the origin (e.g. `(x**2 + y**2)**(1/2)`)
  - Use at least **three** assertions in your unittest method `TestPoint.test_eq()`:
    - Use `self.assertEqual()` to confirm that 2 points with the same distance from origin **and the same x and y attributes** compare as equal
    - Use `self.assertEqual()` to confirm that 2 points with the same distance from origin **but different x and y attributes** compare as equal
    - Use `self.assertNotEqual()` to confirm that 2 points with different distances from origin do not compare as equal
- `Point.within(self, distance, other)`
  - returns `True` iff `self` and `other` are within `distance` from each other
  - As above, use the pythagorean theorem to find the distance between two points; i.e. $sqrt((x2 - x1)^2 + (y2 - y1)^2)$
  - Use at least 2 assertions. As above, `self.assertEqual()` for two points that *are* within some distance of each other and `self.assertNotEqual()` for two points that *are not* within some distance of each other

Reminders

Do not import any modules except for those you wrote yourself (e.g. `Point.py`) and:

- unittest

If you are using the same point in multiple unittests, attach them to your `unittest.TestCase` object in the `setUp` method, as shown above. This method is run at the start of every unittest.

Every method should have a docstring, including unittests.

Grading

Gradescope only looks at functionality here, but the goal of this lab is to get comfortable writing tests. Show your TA your tests as you go, and they'll give you feedback on whether they're good or not.

To earn participation credit on this lab, you need to show your TA your full suite of test cases before leaving lab.

Submitting

At a minimum, submit the following files:

- `Point.py`
- `TestPoint.py`

Students must submit **individually** by the due date (typically, Sunday at 11:59 pm EST) to receive credit.