Mod 2 Homework: OOP and TDD

Test-Driven Development

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

1. Write a test for some functionality you want to implement. Run your code to verify that the test case fails.
2. Modify your code until the test passes
3. Refactor your code: extract duplicate algorithms/classes into a parent function/superclass.

There is an important subtlety in step 1: it is imperative that you run your test after writing it. This ensures that you are writing good tests. If you have yet to implement a certain functionality, but your test still passes, then it is a bad test. This happens more frequently than you may imagine. Start refining your test-creation skills now; it is one of the most important skillsets you can develop in this course.

We will implement rectangular and polar vector classes in this assignment. The steps below walk through the red-green-refactor phases of implmenting the `init` function for a simple vector class - follow along to make sure you understand, then move on to using TDD to develop `RecVec` and `PolVec` classes.

1) Red

Start by adding the following skeleton code to `Vector.py`.

`Vector.py`

```python
class Vector:
    def __init__(self, x, y):
        pass
```

Then, write a test for the `init` function in `TestVector.py`.

`TestVector.py`

```python
import unittest
from Vector import Vector

class TestVector(unittest.TestCase):
    def test_init(self):
        v1 = Vector(3, 4)
        self.assertEqual(v1.x, 3)
        self.assertEqual(v1.y, 4)

unittest.main()
```

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

Terminal

```
$ python .\TestVector.py
E
======================================================================
ERROR: test_init (__main__.TestVector)
----------------------------------------------------------------------
Traceback (most recent call last):
  File ".\TestVector.py", line 8, in test_init
    self.assertEqual(v1.x, 3)
AttributeError: 'Vector' object has no attribute 'x'

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

FAILED (errors=1)
```

Our test fails, which means it is a good test: we know for a fact it tests something our code does not do at present, and if we pass this test later on, we know it is because we have changed the code. Re-read the previous sentence until you parse it - it is subtle and extremely important, and you will save a lot of time if you always verify your test cases fail before implementing functionality.

2) Green

Modify your code until it passes the test case.

```
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

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

OK
```

3) Refactor

In this case, we have nothing to refactor - we have only written one function. Next, it's time to pick the next piece of funcitonality and start from the beginning.

`Vector`, `RecVec`, and `PolVec`

Use TDD to implement three classes: a `Vector` parent class and classes `RecVec` and `PolVec` that inherit from `Vector`. You will be graded on your unittests: you should have one unittest function per attribute.

Remember, write your tests, verify they fail, then add the functionality. The goal is to learn TDD, not to implement vector classes you could copy/paste from StackOverflow in 5 minutes. You will get out of this assignment only what you put into it.

`Vector`

Magic Methods

- `init`
  - explicitly raise a `NotImplementedError` here. Users should specify `RecVec` or `PolVec` instead.
- `add`
  - returns the sum of two vectors
- `sub`
  - returns the difference of two vectors
- `eq`
  - two vectors are equal if their `x` and `y` attributes are equal within 3 decimal points.
  - a `RecVec` and a `PolVec` **can be** equal

The types of objects returned by `add`, `sub`, and `mul` should be determined by the type of the first parameter:

```
>>> r1 = RecVec(3, 4)
>>> p1 = PolVec(5, 30)
>>> isinstance((r1 + p1), RecVec)
True
>>> isinstance((p1 + r1), PolVec)
True
```

Non-magic Methods

- `rectangular`
  - returns the rectangular equivalent of this vector.
  - The returned object should be of type `RecVec` (see below)
- `polar`
  - returns the polar equivalent of this vector.
  - The returned object should be of type `PolVec` (see below)

`RecVec`

Magic Methods

- `init`
  - initializes a new vector with `x` and `y` attributes specified by the user
- `str`
  - returns a string representation of a vector "*RecVec(x = {}, y = {})*" where the curly braces should be replaced with the `x` and `y` attributes

Non-magic Methods

- `get_x`
  - returns the x-coordinate of the vector
- `get_y`
  - returns the y-coordinate of the vector
- `get_mag`
  - returns the magnitude of the vector
- `get_ang`
  - returns the angle (in degrees) of the vector

`PolVec`

Magic Methods

- `init`
  - initializes a new vector with `mag` and `angle` attributes specified by the user. `angle` should be in degrees.
- `str`
  - returns a string representation of a vector "*PolVec(mag = {}, angle = {})*" where the curly braces should be replaced with the `mag` and `angle` attributes

Non-magic Methods

- `get_x`
  - returns the x-coordinate of the vector
- `get_y`
  - returns the y-coordinate of the vector
- `get_mag`
  - returns the magnitude of the vector
- `get_ang`
  - returns the angle (in degrees) of the vector

Exceptions and their unittests

- The `Vector` class should raise a `NotImplementedError` exception if someone calls it directly.
- The arithmetic (`add`, `sub`) and comparator (`eq`) methods should raise a `TypeError` exception if both inputs are not `Vector`s.

Include unittests for all exceptions. You will need to use a `with` statement along with `unittest.assertRaises` (documentation):

```python
def test_init_NIE(self):
    with self.assertRaises(NotImplementedError):
        v1 = Vector() # should raise NotImplementedError
```

External Modules

Feel free to use the following attributes (and only these attributes) from the `math` module:

- `sin`, `cos`, `tan`
- `asin`, `acos`, `atan`
- `pi`

What to test?

We want to test the **public interface** of a class - write tests that look like how a user will use the class. Some functions have been factored out to the `Vector` parent class, but users will access them through either a `RecVec` or `PolVec` object. This means your tests for `add`, `sub`, `eq`, `rectangular`, and `polar` should all be in your test classes `TestRecVec` and `TestPolVec`. The only tests you need in `TestVector` are for exceptions:

- `NotImplementedError` in `init`
- `TypeErrors` in arithmetic and comparators

Submitting

At a minimum, submit the following files:

- `Vector.py`
- `TestVector.py`

Students must submit to Mimir individually by the due date (typically, the second Wednesday day after this module opens at 11:59 pm EST) to receive credit.

Grading

Most of your grade will be for unittests in your unittest file. These will be manually graded after the due date.

30 - Functionality in `Vector.py`

- 11 - `RecVec`
- 11 - `PolVec`
- 3 - Equality
- 5 - Exceptions

70 - UnitTests in `TestVector.py`

- 30 - `RecVec`

- 30 - `PolVec`
- 10 - Exceptions

Feedback

If you have any feedback on this assignment, please leave it here.

We check this feedback regularly. It has resulted in:

- A simplified, clear **Submitting** section on all assignments
- A simplified, clear **Grading** section on all assignments
- Clearer instructions on several assignments (particularly in the recursion module)