Objectives

* Establish a systematic method for testing (TDD).
* Use of magic methods.

Due Date

This assignment is due on Sunday, January 28, 2024, by 6:00 pm.

**Remarks:**

* When you are asked to hand in code, you cut-and-paste your code as text and paste into this Word document immediately following the prompt for your code.
* Include with your code several test case examples and your results.
* This lab is cooperative - talk with your partner as you go through, and make sure you are progressing together.

Value

This assignment is worth 10 points.

Activities

# 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**. Note how the test is written before writing the code.
* Green - modify your code until the test passes
* Refactor - extract duplicate algorithms/classes into a parent function/superclass.

# Red

1. Create a file called Point.py and add the following skeleton code:

|  |
| --- |
| **class** Point:  **def** \_\_init\_\_(self, x, y):  **pass** |

1. Then, write a test for the init function in your if \_\_name\_\_ == '\_\_main\_\_' block.

**if** \_\_name\_\_ **==** '\_\_main\_\_':

p1 **=** Point(3, 4)   
 **assert** p1.x **==** 3   
 **assert** p1.y **==** 4

1. Run Point.py, and you should see something like the following:

|  |
| --- |
| $ python3 .\Point.py  Traceback **(**most recent call last**)**:  File "Point.py", line 7, in **<**module**>** assert p1.x == 3  AttributeError: 'Point' object has no attribute 'x' $ |

Our test fails - **it is a good test.** This assert statement checks for functionality we do not currently have. If it passes in the future, then we know it is because of changes we have made to the code.

# Green

1. Modify your code until it passes the test case. You should see no output when you execute the script in terminal - assert statements produce no output when they pass.

class Point:

def \_\_init\_\_(self, x, y):

self.x = x

self.y = y

# Refactor

In this case, we have nothing to refactor - we have only written one function.

That’s it! You have completed one round of TDD. The next step is to pick a new piece of functionality and repeat the process.

# Magic Methods

1. Update the Point object to add the following functionality. Make sure to use methods that begin and end with 2 underscores, like \_\_init\_\_. Some of these functions have purposefully omitted the appropriate “magic method” name. To access a listing of magic methods available for an object, you can print out a directory of the magic methods (e.g., to find the magic methods associated with a list, dir(list)). Alternatively, you can Google search “magic method to do X in python”. If you are having difficulty finding the correct magic method name after looking for it yourself, feel free to ask.

* Support the comparators <, >, and ==
  + p1 is less than p2 if its distance from the origin is less
  + p1 is greater than p2 if its distance from the origin is greater
  + p1 and p2 are equal if they are the same distance from the origin
* str returns a string representation of a Point (e.g., ‘Point(3,4)’)

1. Before you write your code, write at least two test cases for each magic method – one that expects true and one that expects false. For example, both of the following are required to satisfactorily test the less than method:

# Test of the less than functionality

* assert p1 < p2 # expected True
* assert p2 < p1 # expected False

1. Insert the tests you will perform below this line using the assert command. Remember the use of the if \_\_name\_\_ == '\_\_main\_\_' line of code. Begin by stating what p1 and p2 will be. Be sure your insertion is in courier font.
2. Add to your class definition a non-magic method dist\_from\_origin() which will return the distance of the point from the origin. You cannot use the math module – even for the square root. Paste the final full listing of your code below this bullet.

class Point:  
 *'''Class that defines a point in 2D space.'''* def \_\_init\_\_(self, x, y):  
 *'''Assigns a point to this object with coordinates x and y.'''* self.x = x  
 self.y = y  
  
 def \_\_gt\_\_(self, other):  
 *'''Compares one instanciated point to a nother using Greater Than Magic Method.'''* return self.dist\_from\_origin() > other.dist\_from\_origin()  
  
 def \_\_lt\_\_(self, other):  
 *'''Compares one instantiated point to another using the Less Than Magic Method.'''* return self.dist\_from\_origin() < other.dist\_from\_origin()  
  
 def \_\_eq\_\_(self, other):  
 *'''Compares one instantiated point to another using the Equals Magic Method.'''* return self.dist\_from\_origin() == other.dist\_from\_origin()  
  
 def \_\_str\_\_(self):  
 *'''Returns String representation of instantiated point.'''* return f'Point({self.x}, {self.y})'  
  
 def dist\_from\_origin(self):  
 *'''Calculates the straight line distance using Pythagorean Theorem.'''* return ((self.x \*\* 2) + (self.y \*\* 2)) \*\* 0.5  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 # Test points  
 p1 = Point(3, 4) # length = 5  
 p2 = Point(8, 9) # length > 5  
 p3 = Point(-9, 7) # length > 5  
 p4 = Point(-2, 3) # length < 5  
 p5 = Point(9,-7) # length > 5  
  
 # Test initial coordinates  
 assert p1.x == 3  
 assert p1.y == 4  
  
 # Tests for \_\_gt\_\_ method  
 assert p3 > p4 # p3 is greater than p4 (True case)  
 assert not p4 > p3 # p4 is not greater than p3 (False case)  
  
 # Tests for \_\_lt\_\_ method  
 assert p1 < p2 # p1 is less than p2 (True case)  
 assert not p2 < p1 # p2 is not less than p1 (False case)  
  
 # Tests for \_\_eq\_\_ method  
 assert p3 == p5 # p3 is equal to p5 (True case)  
 assert not p1 == p2 # p1 is not equal to p2 (False case)  
  
 # Test \_\_str\_\_ method  
 assert str(p1) == "Point(3, 4)"  
  
 # Test dist\_from\_origin method  
 assert p1.dist\_from\_origin() == 5.0 # p1 length is 5.0  
 assert not p1.dist\_from\_origin() == 6.0 # p1 length is not 6.0

# Examples

Any examples below are intended to be illustrative, not exhaustive. Your code may have bugs even if it behaves as below. Write your own tests and think carefully about edge cases. You should come up with your own test case values and think about how to verify

p1 = Point(3, 4) # dist\_from\_origin() - 5

p2 = Point(3, 4) # dist\_from\_origin() - 5

p3 = Point(4, 3) # dist\_from\_origin() - 5

p4 = Point(0, 1) # dist\_from\_origin() - 1

p1 > p4 # expected True

p4 > p1 # expected False

p1 > p3 # expected False. Same magnitude,   
 # different coordinates

p1 == p3 # expected True

str(p1)

p1.dist\_from\_origin()

>>> True

>>> False

>>> False

>>> True

>>> Point(3, 4)

>>> 5.0