CPE 101: Fundamental of Computer Science I

(LAB-8)

Due: 5/8/18

For this lab you will explore compound data types (objects) and file I/O.

Download lab8.zip, place it in your `CPE101` directory, and unzip the file. This file includes four subdirectories (corresponding to the different parts of the lab).

unzip Lab8.zip

Functions on Structured Data

This part will be executed with: **python funcs_objects_tests.py**

Objects

Define a class to represent a two-dimensional point. In the `funcs_objects` directory, create a file named `objects.py`. You will define a class in `objects.py` to represent the structure of `Point` objects. The `class Point` type will need two attributes (named `x` and `y`); as before, this means that the `__init__` function must take these two arguments (in addition to `self`) and initialize the attributes within `self`.

Define a class to represent a circle. Add this class to the `objects.py` file. The `class Circle` type will need two attributes to represent the center point (this will be an object of the `Point` class) and the radius.

Be sure to test your `__init__` functions by creating objects and verifying that the attributes have been properly initialized. You can place the test cases in the provided `funcs_objects_tests.py` file.

Note that testing the fields of an object that are themselves objects requires a bit more work than one might initially expect. For instance, when verifying that a `Circle` has been properly initialized, you should not compare the `center` to another `Point`, but should instead compare each field of the `center` point (i.e., the `center.x` and `center.y` components) to the expected values. Comparing objects directly *can* be done, but doing so is beyond the scope of this lab.

Functions on `Point` and `Circle`

In the `funcs_objects` directory create a file named `funcs_objects.py`. Place your test cases in the provided `funcs_objects_tests.py` file.

You must provide at least two test cases for each of these functions. In order to test these functions, you will first need to create an appropriate number of objects and then call the function that you wish to test.

distance

Write a function, named distance, that takes two arguments of type `Point` and that returns the Euclidean distance between these two points.

circles_overlap

Write a function, named circles_overlap, that takes two arguments of type `Circle` and that returns `True` when the circles overlap and `False` otherwise (consider circles touching at the edge as non-overlapping). You must write this function using a relational operator and without using any sort of conditional.

As a helpful hint, two circles will overlap when the distance between their center points is less than the sum of the radii.

Object Equality

As experienced in the previous section of the lab, checking object equality by comparing each field individually is tedious. Though individual attribute comparisons are necessary to properly test the`__init__` function, once object creation is known to work we would prefer to compare objects for equality in a simpler manner. This can be done by defining the `__eq__` function within a class. We will use a simple definition of `__eq__` to reduce the tedium of writing test cases (more sophisticated checks may be introduced in CPE 102).

When an object is compared to another value using `==`, the default behavior is check if the two values are actually the same object (i.e., this is typically referred to as reference equality because both operands must refer to the same object for the check to return `True`). Instead, if you define the `__eq__` function in the object's class, then that function will be called when `==` is used. As such, the`__eq__` can define what it means for the object to be equal to some other value (e.g., they may be considered equal only when each of their attributes is equal).

`__eq__`

Copy your `objects.py` file into the `object_equality` directory. Modify the definition of the `Point` class to add (a simplified) `__eq__` function. This function will take two arguments: `self` (the target of the call, much like with `__init__`) and `other` (the other

operand). The function must return `True` when the `x` and `y` attributes in `self` are equal to the `x` and `y` attributes in `other`. Your function will assume that `other` has these attributes.

The attributes for a `Point` are typically of a floating point type. As such, you should use the `epsilon_equal` defined in `utility.py` when writing your `__eq__` function. *We will discuss this function in class or lab.*

Tests

Modify `object_equality_tests.py` to test that your implementation of `__eq__` behaves as expected. You can do so by using `assertEqual` to compare two `Point` objects.

__ne__

Note that defining `__eq__` does not change the behavior of the `!=` operator. To do this you must define the `__ne__` function for the class. Doing so is not required for this lab, but you might try defining`__ne__` once you have completed the required portions of the lab.

List Comprehensions and Objects

Now lets practice working with lists of objects. Copy your updated `objects.py` from the previous part of the lab into the `list_comp` directory. Develop the functions below in the `list_comp` directory in file`list_comp.py`.

Note: You must implement these functions using a **list comprehension**.

distance_all

Using the map pattern we learned in Lab 5, the `distance_all` function computes and returns a list containing the distance from the origin to the corresponding point in the input list. You may use your `distance` function from the first part of this lab if you like.

are_in_first_quadrant

Using the filter pattern we learned in Lab 5, the `are_in_first_quadrant` function returns all points in the input list that are in the first quadrant (i.e., both x- and y-components are positive) of the Cartesian plane.

Testing

Write two tests for each function above in `list_comp_tests.py`

File Input

Develop the following in the `file` directory in `file.py`.

Write a program that opens a file named `in.txt` and reads each line. For each line in the file, print the line number, the number of characters in the line, and the line itself. For example, given the following `in.txt` file:

```
I am a file.
This is a line.
This is the last line.
```

The output would be:

```
Line 0 (12 chars): I am a file.
Line 1 (15 chars): This is a line.
Line 2 (22 chars): This is the last line.
```

Demonstration

Demonstrate your tests for the first 3 parts of the lab and a sample run of `file.py` using the given example for `in.txt`. In addition, be prepared to show your instructor the source code for all parts of the lab.

Submission

Demo your work and submit your files in PolyLearn.