Lab 2, CSC 101

For this lab you will explore perhaps the single most important building block in programming: the function. The first part of the lab introduces basic output through the use of `print`. The second part of the lab requires you to implement multiple functions on simple data types (e.g., integers, floating points, booleans, and characters). The final part of the lab requires you to implement multiple functions on compound data types (i.e., objects).

Download lab2.zip, place it in your `cpe101` directory (created as part of lab 1), and unzip the file. This file includes three subdirectories (corresponding to the different parts of the lab).

`print`

For this part of the lab you will experiment a little with `print`. `print` allows your program to display text in the terminal window. At this point in the course, the greatest value of `print` is as a tool to assist with tracing execution and debugging.

You will not write test cases for this part of the lab. I know that sounds ludicrous, but there is a reason for the madness. In short, writing test cases (specifically programmatic unit tests) for functions that use `print` requires more features than we have discussed to this point. Instead of unit tests for functions using `print` people tend to fall back to system tests, but we will not do that just yet.

The following steps should be done within the `printing` directory by editing the provided `printing.py` file. I recommend that you do one step at a time (make the specified change and then run the program) so that you can observe the behavior. Each change should be added (in the order specified) to the `print_examples` function (starting at the point indicated by the comment in the function).

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

- Add `print "Entered print_examples function.\n"`

  This prints a string to the console. The `\n` escape sequence indicates that a newline character is to be printed. (Note that, by default, `print` will print a newline character, so you should now see a blank line after the printed message.)

- Add `print "n:", n`

- Add `n = 99`

- Add `print "n:", n, "\tm:", m`

  The `\t` escape sequence indicates that a tab character is to be printed.

- Add `print "f:", f`

There are a number of more interesting aspects of `print`, but this simple introduction is sufficient for now.

Functions

In the `funcs` directory create a file named `funcs.py`. This part of the lab requires that you implement and test multiple functions. You should develop these functions and their tests one at a time. The function implementations must be placed in `funcs.py`. The test cases will, of course, be placed in the provided `funcs_tests.py`.

You must provide at least two test cases for each of these functions. In addition, you should separate your testing into multiple functions (the file includes stubs for the first two testing functions to get you started).

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

$f(x) = 7x^2 + 2x$

Write a function, named f (a poor name, really, but maps from the context), that corresponds to the stated mathematical definition.

$g(x, y) = x^2 + y^2$

Write a function, named g (again, a poor name), that corresponds to the stated mathematical definition.

hypotenuse

Write a function, named hypotenuse, that takes two arguments corresponding to the lengths of the sides adjacent to the right angle of a right-triangle and that returns the hypotenuse of the triangle. You will need to use the `math.sqrt` function from the math library, so be sure to `import math` at the top of your source file.

is_positive

Write a function, named is_positive, that takes a single number as an argument and that returns `True` when the argument is positive and `False` otherwise. You must write this function using a relational operator and without using any sort of conditional (i.e., `if`); the solution without a conditional is actually much simpler than one with. Your test cases should use `assertTrue` and `assertFalse` as appropriate.

Functions on Structured Data

This part of the lab is similar to the previous part, but the functions for this part will work (at least in part) on values of structured data (as represented by objects).

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`. As in the previous lab, 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. If you wish to use any of the functions from the previous part, you will find that the simplest approach at this time is to copy your `funcs.py` file to this directory. To do so, from within the `funcs_objects` directory, type **cp ../funcs/funcs.py .** (the . (dot) at the end of that command is important; it signifies the current directory as the destination of the copy).

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. If an object is to be returned, then you will need to assign the result to a variable and test each field independently.

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.

Handin

You will need to submit your lab code on PolyLearn.