CS5001-5003, Homework 3, Spring 2022

A Two-Week Assignment

There is 1 question with multiple parts. Please submit 1 file for the entire question, called **triangles.py**. This homework should be submitted on Canvas. You should not use *for loops*, *tuples*, *lists*, *arrays*, *objects*, external libraries (except where mentioned here), nor other Python features not yet studied in 5001. Do not confuse "invoking a function with arguments" with "getting keyboard input from the user." Also, do not confuse "returning a value from a function" with "printing a value." In addition to correct I/O behavior, PEP8 style guidelines will be a factor in grading, such as avoiding line length > 79 characters and leaving 2 blank lines between function definitions. Thorough testing is a factor in grading.

1. Triangles [100 points]

In mathematics, a triangle is defined by 3 points; for the purposes of this assignment, a triangle is defined by 6 numbers, representing the (x, y) coordinates of each of the three points. Create a file called **triangles.py**. Start from a copy of our standard template.py file. Divide your work into multiple functions – each function having *one job* -- with a main() function that *tests* your other functions on several examples.

A. [10 points] The function **distance**(x1, y1, x2, y2) should accept as its arguments the coordinates of two points, each an int or a float. It should return the distance between the two points as a (non-negative) float. To take the square root of a number, raise it to the 0.5 power (e.g., 9**0.5 == 3.0); no libraries need to be imported.

Hint: https://www.cut-the-knot.org/pythagoras/DistanceFormula.shtml

Examples:

```
>>> distance(0, 0, 4, 4)
5.656854249492381
>>> distance(0, 2, 2, 0)
2.8284271247461903
```

B. [10 points] The function **perimeter**(x1, y1, x2, y2, x3, y3) should accept as arguments the coordinates of 3 points, each an int or a float. (Three points define a triangle.) This function should return the perimeter of the triangle, as a float.

Examples:

```
>>> perimeter(0, 0, 0, 2, 2, 2)
6.82842712474619
>>> perimeter(0,0, 6, 0, 3, 5.196152422706632)
18.0
```

C. [10 points] The function **area**(x1, y1, x2, y2, x3, y3) should accept as arguments the coordinates of 3 points, each an int or a float. It should return the triangle's area, a float.

Hint: (Heron's formula) *https://www.cuemath.com/measurement/area-of-triangle-with-3-sides/*

Examples:

```
>>> area(0, 0, 3, 0, 0, 4)
6.0
>>> area(-3, 0, 3, 0, 0, 4)
12.0
```

D. [10 points] The function **approx_equal**(d1, d2) should accept as arguments 2 ints or floats. It should return boolean **True** if the two values are within epsilon apart, where epsilon is an identifier *defined in the* ***enclosing scope*** with value 0.001. For example, 2.0 is approximately equal to 2.0001. Otherwise, this function should return **False**.

Examples:

```
>>> approx_equal(1, 1.001)
    True
>>> approx_equal(5.7, 5.71)
    False
```

E. [10 points] The function **approx_isosceles**(x1, y1, x2, y2, x3, y3) should accept as arguments the coordinates of three points, each ints or floats. It should return boolean **True** if any 2 of the sides of the triangle are approximately equal, else **False**.

Examples:

```
>>> approx_isosceles(0, 0, 1, 0, 1, 1.0005)
    True
>>> approx_isosceles(0, 0, 1, 0, 1, 0.95)
    False
```

F. [10 points] The function **approx_equilateral**(x1, y1, x2, y2, x3, y3) should accept as arguments the coordinates of 3 points, each floats. It should return boolean **True** if all 3 of the sides of the represented triangle are approximately equal, else **False**.

Examples:

```
>>> approx_equilateral(1, 3, -2.0, 7, 2.964, 7.598)
    True
>>> approx_equilateral(3, 1, 1, 0, 1, 0.95)
    False
```

G. [25 points] Your **main**() function should ***test*** each of the above functions, with *at least 3 examples* for each function, comparing the actual result returned by your function to the expect result. Example coverage should include potentially unusual cases, to ensure the results are still reasonable. Tests should print the name of the function, its arguments, the expected result, and the actual value. Within main, use an integer variable to keep track of the number of tests failed. main() should print how many tests failed, after running all the tests. To verify correct operation of your test suite, deliberately add one extra copy of one of the tests, but with a wrong expected value, just to demonstrate that a function returning a wrong value would indeed fail.

Examples:

```
>>> main()
    Test: approx_equilateral(1, 3, -2.0, 7, 2.964, 7.598)
    Expected: True, Actual: True
    Failed: 0

    Test: approx_equilateral(1, 3, -2.0, 7, 2.964, 7.598)
    Expected: False, Actual: True
    Failed: 1

    # etc.
    Total Failed: 1
```

In addition to your own 3 test examples per function, also create test cases for this degenerate triangle:

x1 = 0, y1 = 0, x2 = 100, y2 = 100, x3 = 0, y3 = 0

What is the **area** of this "triangle"? What is its **perimeter**? Is it **isosceles**? (etc.)

H. [10 points] **Drawing Triangles**

At the top of triangle.py, after the doc string, import the turtle package, as follows:

```
from turtle import *
```

Be sure to leave appropriate blank lines including two blank lines before the first function definition. Add a new function **draw_triangle**(x1, y1, x2, y2, x3, y3). This should first check whether any of the coordinates are too big for the screen. In that case, it should draw nothing and **return False**. Otherwise, it should draw the triangle with the turtle and then **return True**.

*Hints: It is **not** necessary to* explicitly *create a Turtle object to complete this exercise. When you use **from turtle import ***, then the following commands can be called as ordinary functions. (Python provides a default turtle object.)*
***speed(0)** makes the turtle move at maximum speed (without animation)*
***showturtle**() makes the turtle window appear*
***pendown**() and **penup()** control whether the turtle leaves a trail*
***forward(x)** causes the turtle to move x pixels in the current direction*
***right(d)** causes the turtle to turn right d degrees*
***home()** returns the turtle to the origin*
***xcor**() gets the turtle's x-coordinate and **setx**() sets it*
***ycor**() and **sety**() are analogous for y*
***goto**(x, y) goes to a specific location*
***distance(x, y)** returns the distance to a point, in step units*
***towards(x, y)** returns the angle between the line from turtle position to position specified by (x, y), which depends on the turtle's start orientation. For example, from the starting position (facing East), left(towards(100, 100)) would aim towards (100, 100); then forward(distance(100, 100)) would then move to that point.*

*It is considered more elegant, among Turtle Geometry aficionados, to use **relative** turtle movements, such as forward(100), right(90), instead of absolute positioning. These lend themselves to functions that will work in any position, which in turn support advanced graphic capabilities such as fractals.*

See also: https://docs.python.org/3/library/turtle.html

Your main() should draw a turtle picture of each triangle, as it begins to perform the tests for it, provided that its coordinates fall within the screen boundaries. Note that both ints and floats are acceptable arguments for turtle functions. Testing for the accuracy of the turtle drawings is ***not*** expected!

I. [5 points] **Relative Motion and Reusable Triangles**

From IDLE's HELP pulldown menu, run the built-in Turtle Demo. Try the examples (in the resulting Examples menu). Examine the code for a few examples that you like, such as Fractal Curves, the Towers of Hanoi, and Lindenmayer, perhaps.

Write another triangle-drawing function called **equitri**. It should accept 1 argument, which is the "size" (side length) for an equilateral triangle. It should draw the triangle starting from its current location and heading, using *relative* movement commands such as forward() and right(). It should *return the turtle to the same **state** (position, orientation, pen state, color, etc.) it started in*. Now write a function called **cool_tri(counter)**, using a while loop, which calls **equitri** with a slightly larger length argument each time, until it has called it counter times. Also, after each triangle, have the turtle turn a small amount right or left. Explore variations of this type of spiral pattern. Be creative! Anything that involves triangles, while loops, and includes some sort of variation on each pass

through the loop will do! No test code is required for this, so long as your code draws something that you find interesting involving loops that draw triangles. **For this last part, only, it is OK to work with a partner and brainstorm patterns and coding ideas.** If time permits, we will get started with this part during recitation. You might also experiment with pencolor and fillcolor.

Standard Instructions for CS5001-5003 Assignments (Boilerplate):

1. Except for assignments or parts of assignments where Gradescope submission has been specified, please submit your solution, including any supporting files, using Canvas. Navigate to the *Assignments* page, click o*n Start Assignment*, and then click on *File Upload*. (For some problems, *Text Entry* may suffice.) Please be careful to name your files exactly as specified in the assignment. You may submit multiple times prior to the deadline. However, be careful not to overwrite any needed files from previous, partial submissions. One good strategy is to do all the work for the entire assignment in a single folder, compress it to a ".zip" file, and upload that. Then you can confidently replace it with a newer version, if needed.

2. Standard *Academic Integrity* guidance: For Homework Assignments – unlike Lab Assignments -- please do not collaborate with other students, beyond posting generalized questions and answers on Piazza. When in doubt, you can post your question "Only to Instructors" – but please indicate if it is OK for us to share your question and our answer with the whole group. Also, if you rely on sources on the web or resources other than those assigned, please be sure to cite those sources. (In Labs, Pair Programming is often required or encouraged.) Except when Pair Programming is expected on a Lab, you will usually learn more if you solve each problem on your own. Python's built-in documentation features can be very helpful.