Homework Assignment 5

The assignment is due by 11:59PM of the due date. The point value is indicated in square braces next to each problem. Each solution must be the student’s own work. Assistance should only be sought or accepted from the course instructor. Any violation of this rule will be dealt with harshly.

This assignment explores inheritance. As usual, you are graded not only on the correctness of the code, but also on clarity and readability. I will deduct points for not following the guidelines for your class design, poor indentation, poor choice of object names, and lack of documentation. For documentation, use a common sense approach. While I do not expect every line of code to be explained, all code blocks that carry out a significant task should be documented briefly in clear English.

Please read the submission guidelines at the end of this document before you start your work.

Problem 1 [100 points ] Convex Polygons. A convex polygon is a simple polygon whose shape satisfies the property that as we walk around the boundary of the polygon in counter-clockwise order, we always turn left at each vertex. (Another way to define a convex polygon is that for any two points on or within the polygon, the straight line segment connecting those points lies entirely within the polygon.) See the Figure 1 for an example of a polygon that is not convex and one that is convex.

![Convex Polygons](image)

Figure 1: $P$ is a convex polygon but $Q$ is not.

We can think of a convex polygon as a sequence of vertices specified in counter-clockwise order. Each vertex is a point with an $(x, y)$ coordinate. For example, the polygon $P = \{v_0, v_1, v_2, v_3, v_4, v_5, v_6, v_7\}$ shown in Figure 1 has 8 vertices.

In this problem, you are asked to implement a Point class, a ConvPoly class, a Triangle class, an EquiTriangle class, a Rectangle class, and a Square class. Your implementation
of the ConvPoly class will use Point instances. The Triangle and Rectangle classes are sub-classes of ConvPoly, EquiTriangle is a sub-class of Triangle, and Square is a sub-class of Rectangle. Create a module called convpoly.py to contain all these classes. You will be provided a test file called test_convpoly.py to test your implementation of these classes. Further details are provided below.

**Point class.** Instances of this class are points in two-dimensional space. Hence, each instance has an x coordinate and a y coordinate. We first state a few definitions. We first state a few definitions. See Figure 2 for illustrations.

- **Translation:** If a point \((x, y)\) is translated (moved) by an amount \(s\) in the \(x\) direction and an amount \(t\) in the \(y\) direction, its new coordinates are \((x + s, y + t)\).
- **Rotation:** If a point \((x, y)\) is rotated by angle \(\theta\) about the origin, its new coordinates are \((x \cos \theta - y \sin \theta, x \sin \theta + y \cos \theta)\).
- **Distance:** The distance between two points \((x_1, y_1)\) and \((x_2, y_2)\) is \(\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}\).
- **Sidedness:** Let \(p = (p_x, p_y)\), \(q = (q_x, q_y)\), and \(r = (r_x, r_y)\) be three points. We say that \(p\) lies to the left of \(\overrightarrow{qr}\) (that is, the points \(< q, r, p >\) make a left turn) if and only if \((r_x p_y - p_x r_y) + (q_x r_y - q_x p_y) + (q_y p_x - q_y r_x) > 0\). We say that \(p\) lies to the right of \(\overrightarrow{qr}\) (that is, the points \(< q, r, p >\) make a right turn) if and only if \((r_x p_y - p_x r_y) + (q_x r_y - q_x p_y) + (q_y p_x - q_y r_x) < 0\). Finally, \(p, q, \) and \(r\) are collinear (all lie on a straight line) if and only if \((r_x p_y - p_x r_y) + (q_x r_y - q_x p_y) + (q_y p_x - q_y r_x) = 0\).

![Figure 2](image-url) (a) Translation and rotation of a point \(p\). (b) \(p\) lies to the left of \(\overrightarrow{qr}\). (c) \(p\) lies to the right of \(\overrightarrow{qr}\).

Methods for the Point class are described below:

1. \_init\_: The constructor sets the values of the \(x\) and \(y\) coordinates of the point. The default values for the point is \((0, 0)\).
2. translate: Translates the point by \((s, t)\). The instance is modified after this method is called.
3. rotate: Rotates the point by an angle \(\theta\). The instance is modified after this method is called.
4. distance: Returns the distance between the point and another point \(p\). Hence, this method has one parameter, namely a point \(p\).
5. left_of: This method has two parameters \(q\) and \(r\). It returns True if the point lies to the left of \(\overrightarrow{qr}\), and False otherwise.
6. **right_of**: This method has two parameters $q$ and $r$. It returns `True` if the point lies to the right of $\overrightarrow{qr}$, and `False` otherwise.

7. **str**: Returns a string representation of the point. Use the usual notation to represent a point as its $x$ and $y$ coordinates surrounded by parentheses.

8. **repr**: Produces a (printable) representation of the point.

**ConvPoly class**. Instances of this class are convex polygons. Each instance is made up of a sequence of points, which are the vertices of the polygon specified in counter-clockwise order. We need one definition before describing the methods of the `ConvPoly` class:

- A convex polygon $P$ is said to contain a point $q$ if the point $q$ lies inside the polygon. Observe that a point lies inside a convex polygon if and only if it lies to the left of every consecutive pair of vertices in counter-clockwise order about the boundary of the polygon. In other words, if $v_i$ and $v_{i+1}$ are two consecutive vertices of $P$, then $q$ must lie to the left of $\overrightarrow{v_iv_{i+1}}$.

Implement the following methods for the `ConvPoly` class:

1. **init**: The constructor takes an arbitrary number of points as parameters, where the points are the vertices of the polygon listed in counter-clockwise order about the boundary. Store the points in a list. You may assume that the polygon is convex and that the vertices are in counter-clockwise order; your constructor does not have to check for these conditions.

2. **translate**: Translates the convex polygon by $(s, t)$. This is equivalent to translating every vertex of the polygon by $(s, t)$.

3. **rotate**: Rotates the convex polygon by angle $\theta$. This is equivalent to rotating every vertex of the polygon by $\theta$.

4. **contains**: This method has a single parameter, namely a point $p$. It returns `True` if the polygon contains the point $p$ and `False` otherwise. See the explanation above to determine how to implement this method.

5. **iter**: Return an iterator object. (As with other examples seen in class, this would be the convex polygon itself.)

6. **next**: Return the next vertex from the convex polygon. If there are no further vertices, raise the `StopIteration` exception.

7. **len**: This method allows us to use the built-in function `len` with convex polygon instances (just as **str** allows us to use the built-in function `str`). It returns the “length” of the object. In this case, that is simply the number of vertices in the polygon.

8. **getitem**: Overload the index operator. If the parameters of this method are `self` and $i$, it returns the $i$-th vertex of the convex polygon. If the index is out of range (less than zero or greater than or equal to the number of vertices of the polygon), raise the `IndexError` exception.

9. **repr**: Produces a (printable) representation of the polygon. One obvious way to do this is to print the vertices of the polygon (in counter-clockwise order).

10. **perimeter**: Return the perimeter of the convex polygon. **Hint**: Use the distance method for points.

**Triangle class**. Instances of this class are triangles. Since a triangle is a convex polygon, this class is a subclass of the `ConvPoly` class. This class customizes one method (the constructor) and extends one method (area).
1. **__init__**: The constructor takes three parameters (in addition to `self`), each of which is a `Point`. The constructor should first make sure that the three points do not all lie on a straight line (this is done by checking that one point is to the left or to the right of the other two). If the three points lie on a straight line, raise an exception. If they do not, utilize the `ConvPoly` constructor to customize.

2. **area**: Returns the area of the triangle. You may use Heron's formula, which computes the area of a triangle from its edge lengths. If the triangle has edge lengths `a`, `b`, and `c` and `s` is its semi-perimeter (one half of the perimeter), then the area is given by the formula $\sqrt{s(s-a)(s-b)(s-c)}$.

**EquiTriangle class**. Instances of this class are equilateral triangles; i.e., triangles in which all edge lengths are equal. Since an equilateral triangle is a triangle, this class is a subclass of the `Triangle` class. This class customizes one method (the constructor).

1. **__init__**: The constructor takes a single parameter (in addition to `self`), which is the length of the equilateral triangle. An equilateral triangle of that edge length, *with one vertex at the origin*, is created. *Note*: Utilize the `Triangle` constructor to customize.

**Rectangle class**. Instances of this class are rectangles. Since a rectangle is a convex polygon, this class is a subclass of the `ConvPoly` class.

1. **__init__**: The constructor takes two parameters (in addition to `self`), which are the length and width of the rectangle. A rectangle of those dimensions, *with one vertex at the origin*, is created. *Note*: Utilize the `ConvPoly` constructor to customize.

2. **area**: Returns the area of the rectangle.

**Square class**. Instances of this class are squares. Since a square is a rectangle, this class is a subclass of the `Rectangle` class.

1. **__init__**: The constructor takes a single parameter (in addition to `self`), which is the length of the square. *Note*: Utilize the `Rectangle` constructor to customize.

**Submission Guidelines**

Please name the module `convpoly.py`, as specified above. Make sure that your name and RUID appear as a comment at the very top of the file.

Submit your homework files via Sakai as follows:

1. Use your web browser to go to the website sakai.rutgers.edu.

2. Log in by using your Rutgers login id and password, and click on the OBJECT-ORIENTED PROG F18 tab.

3. Click on the 'Assignments' link on the left and go to 'Programming Assignment #5’ to find the homework file (hw5.pdf). I will also provide a test file for your class implementations in a few days.

4. Use this same link to upload your homework file (`convpoly.py`) when you are ready to submit.

You must submit your assignment at or before 11:55PM on December 12, 2018.