Object-Oriented Programming 50:198:113 (Spring 2019)

Homework:	**6**	**Professor:**	**Suneeta Ramaswami**
Due Date:	**5/6/19**	**E-mail:**	`suneeta.ramaswami@rutgers.edu`
Office:	**323 BSB**	**URL:**	`http://crab.rutgers.edu/~rsuneeta`
		Phone:	**(856)-225-6439**

Homework Assignment 6

The assignment is due by 11:55PM 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 be sought or accepted only from the course instructor. Any violation of this rule will be dealt with harshly.

This homework assignment is on inheritance. Keep in mind that when implementing methods of a subclass, you must reuse (to the extent possible) methods in the superclass by making appropriate calls to methods in the namespace of the superclass. *You will be penalized for needless repetition of code.*

In Problem 1, you are asked to implement a class for **quadratic equations** as a subclass of polynomials (implemented in Homework #5). In Problem 2, you are asked to implement a hierarchy or classes and subclasses. 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.

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

Problem 1 [25 points] Quadratic Equations. A *quadratic expression* of one variable is an expression of the form ax^2+bx+c, where a, b, and c are real numbers, called the *coefficients*. In particular, a is called the quadratic coefficient, b is called the linear coefficient, and c is called the constant coefficient. Since a quadratic expression is simply a polynomial of degree 2, in this problem you are asked to create a `Quadratic` class *as a subclass* of the `Polynomial` class. Implement your `Quadratic` class in a module called `quad.py`, which imports the `Polynomial` class from the module `poly.py` I have also provided a test module called `test_quad.py` (which imports your `quad.py` module). Simply type `python3 test_quad.py` to test your implementation of the `Quadratic` class.

Note: If your implementation of the `Polynomial` class in Homework #5 is fully correct, you may use your own `poly.py` module. If not, please download my solution (in the `Solutions` folder under `Resources`), rename it as `poly.py` and use that module instead.

For a quadratic expression, a *real root* is any real value x such that $ax^2 + bx + c = 0$. In other words, a real root is a value of x for which the quadratic expression evaluates to 0. For example, $2x^2 - 5x + 2$ has two real roots: $x = 2.0$ and $x = 0.5$ because $2x^2 - 5x + 2$ evaluates to 0 at both those values of x. The real roots of any quadratic expression can be found by applying the following rules:

1. If a, b, and c are all zero, then every value of x is a real root.
2. If a and b are zero, but c is nonzero, then there are no real roots.
3. If a is zero and b is nonzero, then there is only one real root, which is $x = -c/b$.

4. If a is nonzero and $b^2 < 4ac$, then there are no real roots.
5. If a is nonzero and $b^2 = 4ac$, there is one real root, which is $x = -b/(2a)$.
6. If a is nonzero and $b^2 > 4ac$, there are two real roots, which are $x = \frac{-b+\sqrt{b^2-4ac}}{2a}$ and $x = \frac{-b-\sqrt{b^2-4ac}}{2a}$.

In this problem, you are asked to implement a `Quadratic` class as a sub-class of the `Polynomial` class. We first describe below those methods of the `Polynomial` class that are *customized* for the `Quadratic` class. Next, we describe methods that *extend* the `Quadratic` class. Keep in mind that you should reuse methods of the `Polynomial` class whenever possible when customizing (or extending) methods for the `Quadratic` class.

Customize: The following methods of the `Polynomial` class are to be customized for the `Quadratic` class.

- `__init__`: Since a quadratic expression is always of the form $ax^2 + bx + c$, the parameter list for the constructor could consist simply of the quadratic, linear, and constant coefficients. Hence, the constructor for the `Quadratic` class has three real numbers as parameters, which are the quadratic, linear, and constant coefficients, in that order. As shown in the examples in class, in order to avoid unnecessary code repetition, the constructor should be implemented by calling the `Polynomial` class constructor with the suitable parameter list. Hence, an example of creating a `Quadratic` instance for the quadratic equation $2x^2 - 5x + 2$ would be as follows:
  ```
  Q = Quadratic(2, -5, 2)
  ```
- `addterm`: We need to customize this method for `Quadratic` objects because only terms with exponents 2, 1, or 0 can be added to a `Quadratic` object in order to maintain it as a `Quadratic` (adding a term with higher exponent would imply that the object is not a quadratic equation any more). Therefore, this method must first check that the exponent of the term being added is legal. If it is not, an exception should be raised. If it is, then the term may be added. Once again, call the `Polynomial` method suitably instead of repeating code.
- `__add__`: Since the sum of two quadratics is also a quadratic, the customization required here is that the `__add__` method for `Quadratic` objects should return a `Quadratic` as well. Implement this method to ensure that a `Quadratic` object is returned.
- `__sub__`: For the same reasons described above in the `__add__` method, you will need to customize the `__sub__` method as well so that it returns a `Quadratic`.
- `__mul__`: When two quadratics are multiplied, the result may or may not be a quadratic. Customize the __mul__ method for the `Quadratic` class so that it returns a `Quadratic` object if the degree of the result is at most 2, and a `Polynomial` object otherwise.
- `scale`: Once again, since the result of scaling a quadratic is a quadratic, you must customize the `scale` method for the `Quadratic` class so that it returns a `Quadratic`.

Extend: The following method is specific to quadratic equations, and hence we extend the `Quadratic` class with this method.

- `roots`: This method returns *a list* containing the real roots of the quadratic equation. If the quadratic equation has no real roots, an empty list is returned. If the quadratic equation has one real root, a list containing that single root is returned.

If the quadratic equation has two real roots, a list containing both roots is returned. Finally, if the quadratic equation has infinitely many roots (rule #1 above), a list of length 3, containing all zeros, is returned.

Problem 2 [50 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.

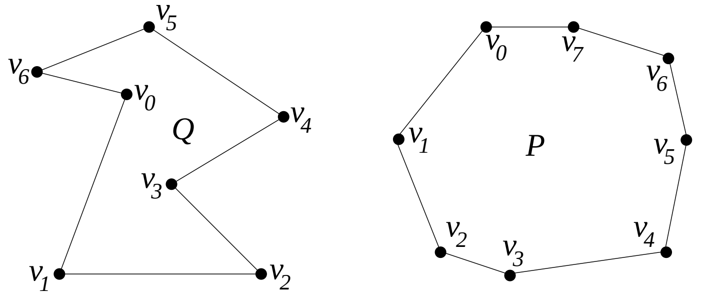

Figure 1: P is a convex polygon but Q is not.

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 θ about the origin, its new coordinates are $(x \cos \theta - ysin\theta, x \sin \theta + y \cos \theta)$.
- **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

$(r_xp_y - p_xr_y) + (q_xr_y - q_xp_y) + (q_yp_x - q_yr_x) < 0$. Finally, p, q, and r are collinear (all lie on a straight line) if and only if $(r_xp_y - p_xr_y) + (q_xr_y - q_xp_y) + (q_yp_x - q_yr_x) = 0$.

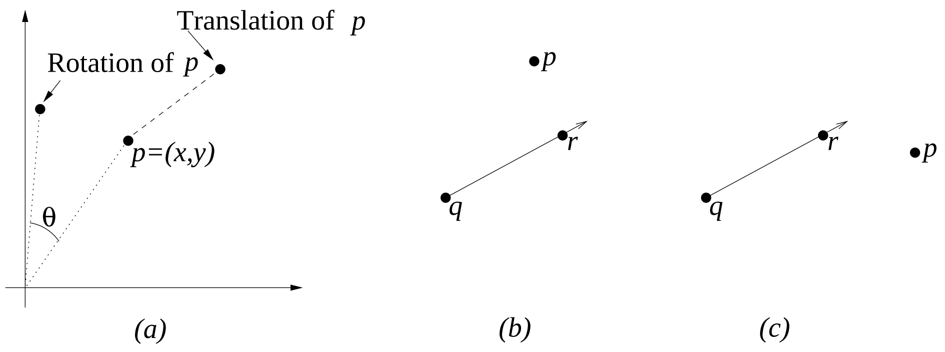

Figure 2: (a) Translation and rotation of a point p. (b) p lies to the left of $\overrightarrow{qr}$. (c) p lies to the right of $\overrightarrow{qr}$.

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 θ. 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 $\overrightarrow{\texttt{qr}}$, 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 $\overrightarrow{\texttt{qr}}$, 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 θ. This is equivalent to rotating every vertex of the polygon by θ.
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 for the convex polygon.
6. `__next__`: Return the next vertex from the convex polygon. If there are no further vertices, raise the `StopIteration` exception. **Note:** If your implementation of the `__iter__` method returns a new iterator object, then the `__next__` method will be a method of that class.
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 edges 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

Implement the first problem in a module called `quad.py` and the second problem in `convpoly.py`. **Your name and RUID should appear as a comment at the very top of each module.** Test each of your programs thoroughly before submitting your homework. When you are ready to submit, upload your files on 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 S17'` tab.
3. Click on the 'Assignments' link on the left and go to 'Homework Assignment #6' to find the homework file (`hw6.pdf`), and the test modules `test_quad.py` and `test_convpoly.py`.
4. Use this same link to upload your two homework files (`quad.py` and `convpoly.py`). Test your modules thoroughly before submitting.

You must submit your assignment at or before 11:55PM on May 6, 2019.