Problem 2: Millisoft "Shapes" Revisited (35 Points; individual only)

Copied from:
https://www.cs.hmc.edu/twiki/bin/view/CS5/MillisoftShapeBlack
on 3/22/2017

This is the sole **individual only** problem this week.

The objective of this problem is get into "shape" as object-oriented designers and programmers.

Shape Up!

Begin by downloading and unzipping the code that we discussed/developed in class:

- hw10pr2.zip

The `turtle` package is fully documented on the Python turtle package site.

Within that zip file you should find three files:

- Vector.py
- Matrix.py
- Shapes.py

Now, modify the `Shapes.py` file as follows:

- Note that there is an error in the render function of the `Circle` class! In particular, the way that the turtle package renders a circle of a given diameter is not with the circle centered at the turtle's location but rather with the center located `radius` units left of the turtle. You can see this by just importing the turtle package in the Python interpreter and drawing a few turtles of different radii. Change the circle's `render` method so that the circle is actually rendered with its center at `self.center`.
- Add another geometric shape of your choosing (e.g. triangle, polygon, etc.). Include a docstring that explains the arguments to the constructor (that is, the `__init__` function).
- Add a method called `translate` that takes a Vector as input and translates *any* of your current shapes by this Vector. To get the most

out of inheritance, this function should go in the `Shape` class. Any specific shape that can't use the general form of `translate` in the `Shape` class can override that general version of `translate` with its own version.

- Modify the `rotate` method (again, in the `Shape` class with special cases in other shapes only where necessary) so that it has the following arguments:
    - `theta`: The rotation angle, in degrees.
    - `rotateAbout`: A vector (you can think of it as a point) such that the shape will be rotated about this point. The default value of this variable should be the origin (0, 0). **Recall** that the rotation algorithm that we saw in class—using a rotation matrix—rotates a point counter-clockwise around the origin. To rotate about the point `rotateAbout` with coordinates `(rotateAbout.x, rotateAbout.y)`, we first imagine *translating* the point `(rotateAbout.x, rotateAbout.y)` so that it coincides with the origin. In reality that means that every point that we wish to rotate has to be translated by that same amount—namely by `-rotateAbout.x` in the x-dimension and `-rotateAbout.y` in the y-dimension. Now, we can rotate our newly translated point about the origin. Finally, after rotating that point, we "undo" the translation by translating back to its original frame of reference—namely by translating by `rotateAbout.x` in the x-dimension and by `rotateAbout.y` in the y-dimension. Basically, the idea here is to change the frame of reference so that we can rotate about the origin (which we can do with matrix multiplication) and then change the frame of reference back to its original state.
- Add a method called `scale` that takes a single floating point number `s` as input and scales *any* of your current shapes by a factor of `s` about its center. That is, the center of the new scaled shape should be the same as the center of the original shape! (Translation will be required to make this work just like you did with rotation. Our scaling algorithm using matrices depended on the object being centered at the origin. So, we'll translate to the origin, scale, and translate back.) You should use matrix multiplication here to do the actualy scaling (but not the translation) of the points in the shape. `scale` should be defined in the `Shape` class and only overridden in specific shapes where necessary.
- Write a `flip` method that flips any shape with respect to a given line. The line is specified by passing `flip` two inputs, each of which is a `Vector`. These two vectors represent two points and thus define a line (the line used for the flip). This function should be defined in the `Shape` class and only overridden for specific shapes where

necessary. **Note:**Flipping about an arbitrary line may seem hard. Notice though that flipping about one of the axes (x- or y-axis) is easy! So, consider transforming your flip line so that it coincides with one of the two axes. What that *really* means is transforming (translating and rotating) your points by that "amount". Then, you can flip about the axis that you chose. Then you can "undo" the transformation to bring your points back to their original frame of reference! (You might need to do a little trigonometry here.)

- **For +5 Optional Bonus Points...** Snoop around on the web or in a linear algebra book to learn how to compute the area of any 2D polygon (assuming that its edges don't intersect one another and that its vertices are given in counter-clockwise orientation) and add an `area()` function to your `Shape` class that returns the area of any 2D polygon (again, you may assume that the vertices are always specified as you walk around the boundary of the polygon counter-clockwise). If you're having difficulty reading in the vertices, make sure to carefully read over the Vector.py file.

Ship Out!

Now, use your modified `Shapes.py` class to write a program in a file called `Pretty.py` that draws a pretty picture that showcases all of the shapes and their methods. The grutors and the profs will be looking forward to ooh-ing and aah-ing over your pictures.

Your program will begin with:

```
from Shapes import *
```

The actual function that draws the picture should be called `main()` and it should be invoked automatically using the "main trick".

Finally, submit your files in a newly-zipped `hw10pr2black.zip` file. This should include

- your `Shapes.py` file,
- your `Vector.py` file,
- your `Matrix.py` file, and
- your `Pretty.py` file.