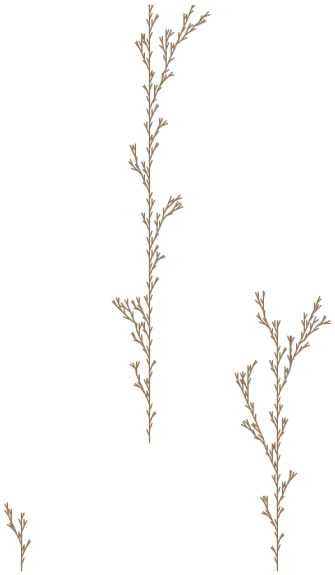
Zena's CS151 Project9

This week's project was all about making shapes using classes, and applying the algorithm of making a shape to making an L system object. Instead of making each shape follow procedures, we only had to change the angle and string to make new shapes based on the "Shape" class; this concept is called "inheritance".

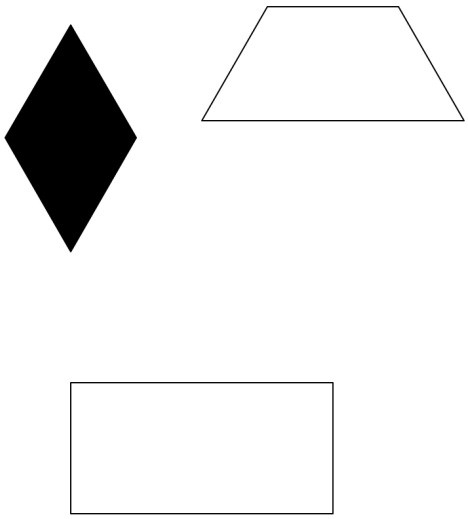
Task 1 was to make the tree L systems draw using the Shape class inheritance. I made a new file that contained a Tree class, and initialized trees by inheriting the Shape class from shapes.py and simply making the text dynamic (able to change) so it could account for the non-static nature of the tree file text. Then, to actually draw a tree, I would have to assign a Tree object to a variable, then call the draw method to draw object in the variable. The code for this part is seen below; first the string is created by calling on the l system file, then the object assigns the string to a field, and then a shape object is made and drawn on the canvas. Since self is a Shape class instance in this case, the string is already passed in when it is stored in self, and the turtle interpreter connected to the Shape class can read the string in and decide how to draw it. The trees are also shown below.

```
x=self.lsystem.buildString(self.iterations)
self.string=x
shapes.Shape.draw(self,xpos,ypos,scale,orientation,iter=self.iterations)
```



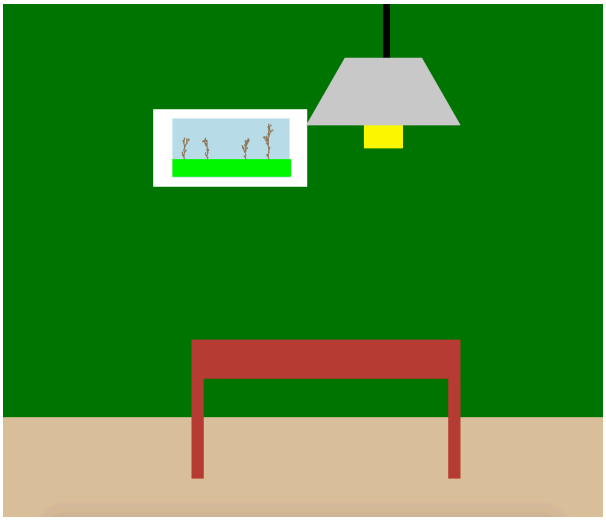
For Task 2, we needed to create at least three additional shapes generated from the Shapes class, in addition to the square and triangle from the lab. I made a hexagon, a diamond/rhombus, and a rectangle. They were all pretty straightforward to make; it was just a matter of inheriting the Shape object then changing the string and the angle. The code to make the diamond, for example, is shown below. As you can see, the angle is set to thirty, and so the turtle first turns left 60 degrees, then moves forward, then turns 60 degrees again, then moves forward, then turns 120 degrees, then moves forward, etc., until it gets back to the original position and orientation. The point at which the diamond starts to be drawn from is the bottommost corner. Making it filled in was just a matter of adding curly brackets to the string and making the turtle interpreter get a new case on what to do with curly brackets.

```
class Diamond(Shape):
    """makes a diamond object by calling upon the Shape class
    and giving specific parameters to the object
    """
    def __init__(self, distance=100,color=(0,0,0)):
        Shape.__init__(self,distance,30,color,'{++F++F++++F++F}')
```



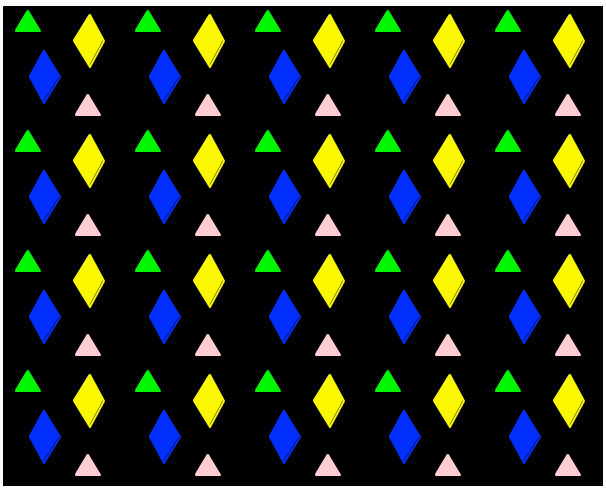
Task 3 was to make an indoor scene composed of various shapes created through inheriting the Shape class properties. This was very simple; all I had to do was create the objects, then draw them at the desired locations. The code to make a table (composed of three rectangles), for example, is just calling various shapes to be created, then to be drawn. Any changeable specifications such as color can be input when creating the shape (such as "color='brown'"). The code and picture are shown below.

```
#draws the top part of the table
tabletop=shapes.AltRectangle2(distance=50,color='brown')
tabletop.draw(-150,-100)
#draws two legs for the table
tableleg=shapes.AltRectangle(distance=15,color='brown')
tableleg.draw(-150,-100)
tableleg.draw(185,-100)
turtle_interpreter.TurtleInterpreter().hold()
```



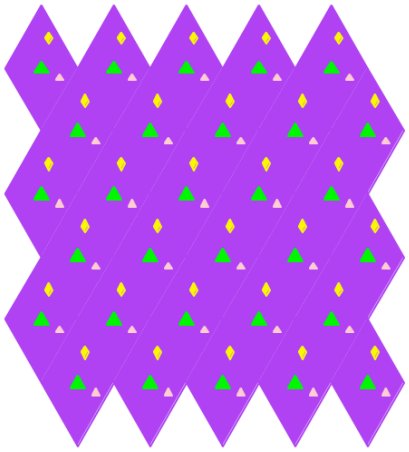
The goal of task 4 was to be able to make an object composed of multiple objects (in this case, a square "tile" containing other shapes) then pasting it in a tiled pattern with specified dimensions. Each "tile" was scale*scale wide, so to prevent them from overlapping, I made a loop that took the current loop variable in the range Nx (the amount of columns, moving on the x axis), and multiplied it by the scale, then added that to the x location of the placement for the tile. The first time, the loop variable would be 0, so the tile would just be in its regular place, with its bottom left corner at the given (x,y) coordinate, but after that, the tiles would start shifting over just enough to prevent overlap. The same idea was used for the y axis. The code showing this concept is below, along with the image.

```
def mosaic(x,y,scale,Nx,Ny):
    for i in range(Nx):
        for j in range(Ny):
            tile(x+scale*i,y+scale*j,scale)
```



For extension 1, I chose to make diamond tiles. First, I assembled a "tile" as I did with the square ones in Task 4. Next, I made a function that would assemble it all into a "mosaic", like before. The content of this function was the major difference between it and the square tile mosaic; for this, moving over by scale times the loop variable would just create overlap, so instead, I had to observe the pattern of my pictured mosaic. For every other row, the tiles would need to shift down half of the tile's height and right half of the tile's width. I used "if i % 2 == 0" to test if the column number was even. If it was, I would draw the tiles shifted half the scale times the loop variable away from the origin; this would not need to change for the odd columns. The thing that needed to change for the odd columns was the tile's vertical position; it just needed to be exactly what I had, except with everything shifted down by half of the height. In this case, half of the height was scale/2 times the square root of 3 (obtained from 30-60-90 triangle rules), and so I just subtracted that from what was already there for y, and the shown image was the result.

```
for i in range(Nx):
    for j in range(Ny):
        if i % 2 == 0:
            exttile(x+scale/2*i,y+scale*math.sqrt(3)*j,scale)
        elif i % 2 != 0:
            exttile(x+scale/2*i,(y+(scale*math.sqrt(3))*j)-scale/2*math.sqrt(3),scale)
```



I also did extension 3, which was to make the branches of the trees droop down. As you can see, the bigger the tree (so the more iterations), the droopier it becomes. This is because I passed in the iterations (as an optional argument) through the tree.py file into the draw function of the shapes file, and from there into the turtle interpreter. If the turtle interpreter had this passed in, it would then consider how many iterations the tree had (I told it, if it is more than two iterations, to really show any significant effect), then loop over the iterations in order to reach each branch. Instead of just going forward, though, it would go forward then turn left a bit (.2*iterations, in this case). This would make everything bend toward the left at an exponential rate with the increasing iterations. The code and image are shown below.

```
if c=='F':
    if iterations>2:
        for i in range(iterations/iterations):
            turtle.forward(distance)
            turtle.left(angle*.02*iterations)
    else:
        turtle.forward(distance)
```

This project taught me a lot about making shapes by inheriting classes, as well as considering shape properties to decide what patterns to use when making repeated instances of them without overlap. It also taught me how to carefully keep track of all the parameters being passed through various classes and into other files; understanding this allowed me to manipulate certain properties of the L systems.

Worked with: Prof. Taylor