Project3

In this project, I used a lot of the code from my previous project(project2) and worked to make the code shorter and more efficient by using loops, lists, and conditionals. Using lists and for loops allowed my code to complete the same task with less lines and more organization. Also, conditional statements in code is something new I learned that makes the function complete one of a number of tasks depending on an input. For example, using "if" and "else" statement. This project focused on making the function coding in Project two efficient by making code more organized, more clear, and shorter with lists, loops, conditionals, and annotations.

First off, in **task 1** I created a new file titled better_shapelib.py and copied most my code from shapelib.py in project 2. Then, I was asked to edit both my goto() function and buildingblocks() function in shapelib.py by adding a new parameter called "fill." Next, I had to edit my code for my buildingblocks() function by adding conditionals. Basically, if the "fill" parameter has the value "True" than the block drawn should be filled with color and if the "fill" parameter has the value "False" than the block drawn should not be filled. To do this I used if and else conditional statements. My code for **task 1** looked like this:

```
def block( x, y, width, height, fill, color ):
    goto( x, y )
    print 'block(): drawing block of size', width, height
    # if the parameter fill is true then do this
    if fill == "True":
        t.begin_fill()
        t.color(color)
        for i in range(2):
            t.forward(width)
            t.left(90)
            t.forward(height)
            t.left(90)
        t.end_fill()
    # if the parameter fill is false then do this
    else:
        for i in range(2):
            t.forward(width)
            t.left(90)
            t.forward(height)
            t.left(90)
```

Second, for **task 2**, I was asked to edit two of my simple shape functions from shapelib.py by using for loops and conditionals to simplify the code. Also, I was told to add "fill" parameters like the ones in task 1 to my simple shape functions. So, I edited my functions for star(), parallelogram(), and cross() with for loops and fill parameters. Here is what the code for the new star() function looked like:

```
def star( x, y, scale, fill, color ):
    '''draws a star given location, scale, and color'''
    goto( x, y )
    if fill == "True":
        t.begin_fill()
        t.color(color)
        for i in range(10):
            t.forward(50*scale)
            t.right(108)
            t.forward(50*scale)
            t.left(144)
        t.end_fill()
    else:
        t.begin_fill()
        for i in range(10):
            t.forward(50*scale)
            t.right(108)
            t.forward(50*scale)
            t.left(144)
```

Task 3 asked to repeat the same process from task 2 but with a couple of my aggregate shapes from project 2. So, I edited my functions for leaf(), and tree() with for loops and fill parameters. Here is what the code for the new star() function looked like:

```
# leaf() function rewritten with multiple argument for loop
def leaf( x, y, scale, fill, color ):
    t.tracer(False)
    '''draws a leaf given location and scale'''
    goto( x, y )
    dists = [ 8.33, 3.33, 15, 5, 15, 15, 5, 15, 3.33, 8.33, 10, 10, 10, 8.33]
    turns = [ -30, 120, -105, 110, -95, 150, -95, 110, -105, 120, 60, 30, 180, 75]
    if fill == "True":
        t.begin_fill()
        t.color(color)
        for i in range( len(dists)):
            t.left( turns[i])
            t.forward( dists[i]*scale)
        t.end_fill()
    else:
        t.begin_fill()
        for i in range( len(dists)):
            t.left( turns[i])
            t.forward( dists[i]*scale)
```

As you can see, I used the practice of lists in this code to shorten my function length and to facilitate the for loop. The command len() basically just uses the length of the list called. Professor Taylor helped me understand how to use multi argument lists with for loops.

Task 4. This task asked to choose one of my two Maine outdoor scenes from project 2 and edit the code and functions to allow the entire scene to be scaled. I chose to edit all my code for my second outdoor scene or outdoors2(). First, I had to go back through all of my code and edit every single function the pertains to the overall function outdoors2(). I went through every function and made sure they had parameters for x location, y location, and scale. Then, I went through all the code in these function and made sure I had "*scale" at every appropriate spot next to coordinates and distance commands (but not for angles). After chugging through all of that, I had one last thing to do. I had to give the overall function outdoors2() the parameters (x, y, scale). Finally, for all the functions called inside the main function outdoors2(x,y,scale), I had to edit the parameters to look like this (x+0*scale, y+0*scale, 1*scale). Basically, what this did was apply the parameters from the main function to the parameters within the inside functions. I then renamed my main function from outdoors2() to myscene2(). Here is what my top level code for myscene2() looked like:

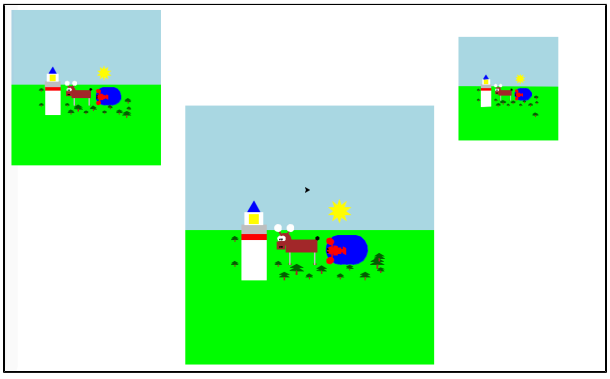
```
# myscene() function takes my second outdoor image from project2
# and allows me to scale and position the entire image
'''This is my new function. It is basically the same as Outdoors2
    but now it allows me to scale and position the entire image'''

def myscene2( x, y, scale ):
    '''scale and position dependent function of outdoors2 in project2'''
    landscape(x+0*scale, y+0*scale, 1*scale)
    lighthouse(x+0*scale, y+0*scale, 1*scale)
    wildlife(x+0*scale, y+0*scale, 1*scale)
```

Next, for **task 5**, I was told to make a new file called taskI.py that imports my better_shapelib.py file as "bsl." Then, in this taskI.py file I was asked to call the scaleable function of my outdoor scene, myscene2(), multiple times. I was asked to draw several different versions of the myscene2() function at different locations and various sizes. Here is what the code looked like:

```
if __name__ == '__main__':
    bsl.myscene2(400,200,.10)
    bsl.goto(0, 0)
    t.setheading(0)
    bsl.myscene2(-450, 200, .15)
    bsl.goto(0, 0)
    t.setheading(0)
    bsl.myscene2( 0, -100, .25)
    bsl.goto(0, 0)
    t.setheading(0)

# used goto() and setheading() to bring turtle cursor back to start point
```

Notice I used the commands goto() and turtle.setheading(). This was to bring the cursor back to the starting point between drawings
This I what the resulting image looked like:

Required Image 1:(varying images)



Task 6. I was told to make a new file called taskII.py that creates a new scene with my old scene somewhere inside of it. For example, a museum. So, I wrote code for a museum scene with my myscene2() image as a painting in the gallery. To do this I obviously had to import my better_shapelib.py file as bsl. In the code for creating the museum scene I used a lot of for loops and lists. Here is some of the code for my museum ceiling:

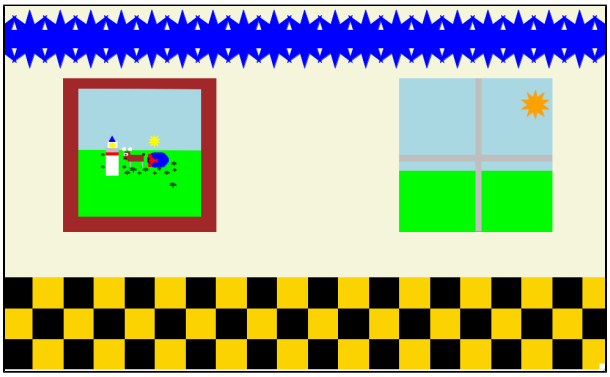
```
def ceiling( x, y, scale ):
    t.tracer(False)
    position = [-600, -550, -500, -450, -400,-350,-300, -250, -200, -150, -100, -50, 0,
                50, 100, 150, 200, 250, 300, 350, 400, 450, 500, 550]
    for i in range(len(position)):
        bsl.star(x+position[i]*scale, y+225*scale, .5*scale, 'True', 'blue')
```

Here is the final code and function for my museum1():

```
def museum1( x, y, scale):
    wall(x+0*scale, y+0*scale, 1*scale)
    ceiling(x+0*scale, y+0*scale, 1*scale)
    floor(x+0*scale, y+0*scale, 1*scale)
    painting(x+0*scale, y+0*scale, 1*scale)
    window(x+0*scale, y+0*scale, 1*scale)
    outsidewindow(x+0*scale, y+0*scale, 1*scale)
    windowpane(x+0*scale, y+0*scale, 1*scale)
```

Here is what the final image of my museum scene looked like:

Require Image 2:(museum)

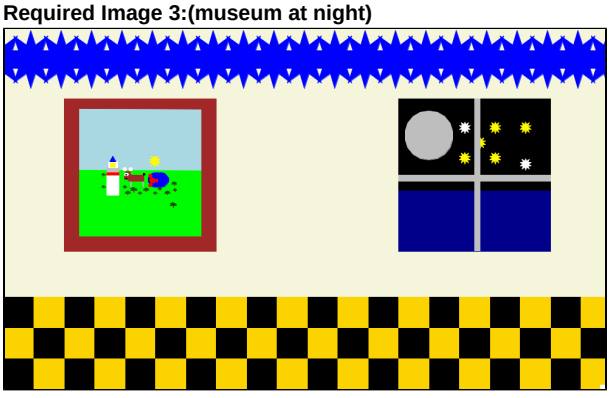


Finally, **task 7**. This task just asked my to edit my museum scene so that it depends on a new parameter. This parameter was asked to be a command-line parameter. What that means it that the parameter is typed in after and right next to where you type the file name in the terminal. I decided to make my changing parameter morning and night. In my museum scene there is a window. In museum1() it is day time. For **task 7** I coded two new functions called outsidewindowmorning() and outsidewindownight(). Then, I imported 'sys' which is something we learned about in lab. Sys, from what we know about it in CS151, can be used to assign command-line parameters. Here is what my final code for museum2() looked like:

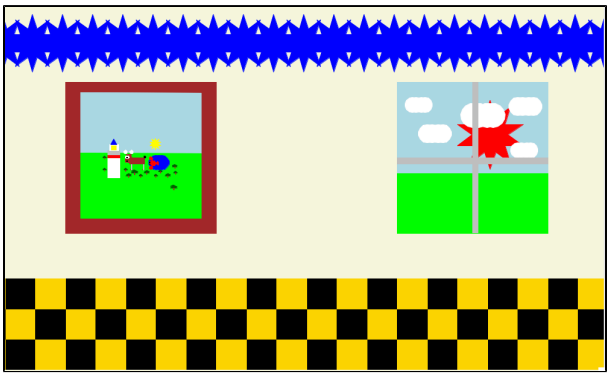
```
def museum2( x, y, scale, time):
    wall(x+0*scale, y+0*scale, 1*scale)
    ceiling(x+0*scale, y+0*scale, 1*scale)
    floor(x+0*scale, y+0*scale, 1*scale)
    painting(x+0*scale, y+0*scale, 1*scale)
    window(x+0*scale, y+0*scale, 1*scale)
    if time == "Night":
        outsidewindownight(x+0*scale, y+0*scale, 1*scale)
    else:
        outsidewindowmorning(x+0*scale, y+0*scale, 1*scale)
    windowpane(x+0*scale, y+0*scale, 1*scale)

museum2(0,0,1, sys.argv[1])
```

Notice the last parameter in my museum2() function is time. Then, when I call the function by writing museum2(0, 0, 1, sys.argv[1]), I write sys.argv[1] where my time parameter is. What sys.argv[1] does is it tells the program to assign the *second* thing typed in terminal to the time parameter. It is important to use [1] not [2] because the counting starts at 0. So sys.argv[0] is simply the file name you call and sys.argv[1] is whatever you type after the file name. I have conditional statements that say if the word typed after the file name in terminal is "Night" then draw the function outsidewindownight() and then I have else draw the function outsidewindowmorning().
Here is the image I get if I type *"python taskII.py Night"* into terminal:

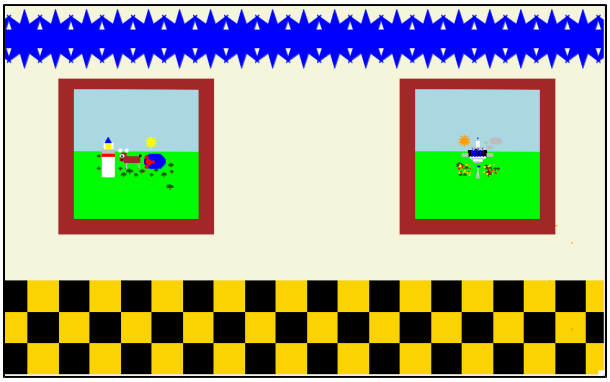
Required Image 3:(museum at night)



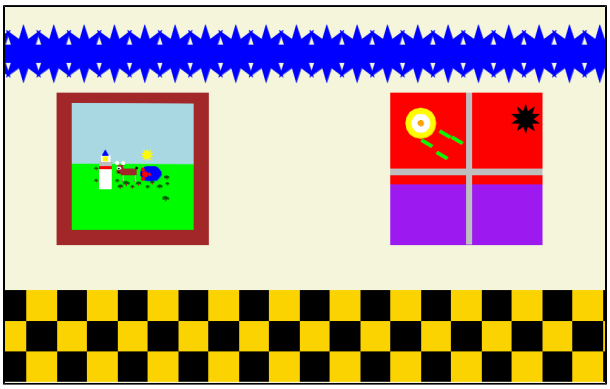
Here is the image I get if i type *"python taskII.py Morning"* into terminal:
Required Image 4:(museum in morning)



After I completed the required tasks for the project, I was encouraged to do further extensions. I decided to do two extensions. My first extension, **extension 1**, was one of the suggested extensions. I decided to add both of my Maine outdoor scenes from project2 to my museum gallery. This required me to go through all of the functions, and the functions within the functions that pertained to the main function outdoors1() and make sure they all contained the correct (x, y, scale) parameters and used *scale in all the appropriate spots. Basically, I repeated what I did for outdoors2() earlier in the project but for my outdoors1() scene. I renamed the function myscene1() and placed it in the museum gallery on the wall next to my other outdoor scene. Here is what the museum gallery fro **extension1** looked like:



Extension 2. For this extension, I built off of my earlier conditional weather code from task 7. I decided to add one more possible outside the window function. This final outside the window function title invasion() illustrates a UFO outside the museum and a red sky and a sun turned black. I made another conditional statement using import sys and argv. If the command-line argument was "Invasion" than the invasion() function will be drawn outside the window. Here is what the invasion image for **extension 2** looked like:



In the end, through this project, I became a more efficient coder and complex coder. I know understand how to use for loops and conditional statements and I have become very proficient in the use of parameters. Also, I learned a lot of new Python skills including importing "sys" and using lists and command-line arguments. All of these things came in handy throughout my project and its extensions. I can make much more complex images with much less code. Being able to use lists to simplify multi-argument code shortened my lengthy and repetitive code and was a vital skill I learned. Overall, this project taught me what organized, more complex but more efficient code looked like.

I received help from Professor Taylor with the use of multi-argument lists with for loops.