6.189 Homework 4

Readings

How To Think Like A Computer Scientist: Wednesday: Make sure you've finished Chapters 12-14 (all), & Chapter 16 (all); Thursday - get all readings finished!

What to turn in

Turn in a printout of your code exercises stapled to your answers to the written exercises by 2:10 PM on Tuesday, January 18th.

Exercise 4.1 – Inventing The Wheel

1. Graphics setup

   - Download the `graphics.py` file from the Materials section of the course webpage. Make sure to save it in the directory you'll be doing your work in.
   - Run the module as if it were a Python program (open it in IDLE and run it). If everything was done correctly you will get a demo window with a triangle and some text. GET AN LA'S HELP if this doesn't work right on your machine!
   - Documentation on the graphics module is on the handout given in class, and also on the Materials section of the course webpage.

2. Basic Graphics Application

   Here is a skeleton of a new graphics program:

```
from graphics import *

#add any functions or classes you might define here

# create a window with width = 700 and height = 500
new_win = GraphWin('Program Name', 700, 500)

# add your code below this point

new_win.mainloop()
```

3. Animating the wheel

   Download the file wheel.py from the Materials section of the course webpage (it is exactly the same as what you saw in class today). Run it and make sure that a wheel appears in a pop-up window.

   Now let's try to add an animate method that would move the wheel across the screen. We will make use of the move method in the wheel class that moves the object dx units in the x direction and dy units in the y direction. Here is what the animate method will look like:

```
from graphics import *

class Wheel:
    ...
    def animate(self, win, dx, dy, n):
        if n > 0:
            self.move(dx, dy)
            win.after(100, self.animate, win, dx, dy, n-1)
```

   The animate method has 4 parameters - a GraphWin object win, the units by which to move the object in the x and y directions, dx and dy, and the number of times to call the animate method, n. The animation will stop when n = 0. The interesting part here is the after method of the GraphWin object. The first parameter is the time in milliseconds after which the GraphWin object will call the animate method again. The second parameter is the function object the GraphWin object needs to call; in our case it is the animate method of the Wheel object. As we mentioned in class, in Python everything is an object - even functions/methods - and they can be passed as parameters to other functions/methods. The rest of the parameters are the new parameters to the animate method. Note that we decrement n by 1 every time we setup a new call to animate.

   Now, write a program that uses the updated Wheel class and create a Wheel object (you can pick the colors[1] of the tire and wheel to be anything you want) and make it move the wheel across the screen by 1 unit in the x direction 100 times. Remember you first need to draw the wheel before you can move it.

Exercise 4.2 – Drawing A Car

1. Drawing Rectangles

   To display a rectangle, you need to specify two points: the upper left corner and the bottom right corner. Remember our y-axis is flipped.

   Make a file car.py and try the code below:

```
from graphics import *

new_win = GraphWin("A Car", 300, 300)

rect = Rectangle( Point( 10,10), Point(200, 100 ) )
rect.setFill( "blue" )
rect.draw( new_win )

new_win.mainloop()
```

   Run your program and make sure that the rectangle appears on the screen.

   Try changing the color and width of the outline of the rectange. Look at the setOutline and setWidth methods.

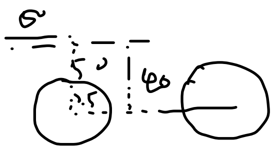
[1]There's a list of available colors in the file rgb.txt on the course webpage.

2. Drawing the car

   In this exercise, we will create a class for a car that will use the `Wheel` class from exercise 1. To do that, include the line `from wheel import *` *directly underneath* the line `from graphics import *`. Be sure that `wheel.py` and `car.py` are saved in the same directory; this will enable you to use your definition of Wheel from 4.1 instead of redefining it. Ask an LA if you're confused about this.

   The car will contain 3 *attributes*: two Wheel objects and one Rectangle object (the body of the car) that is horizontal and whose bottom corners correspond to the centers of the wheels. Below is an example on how to use the `Car` object. Try to figure out how the class `Car` should be defined based on the way it is used.

```
new_win = GraphWin('A Car', 700, 300)

# create a car object
# 1st wheel centered at 50,50 with radius 15
# 2nd wheel centered at 100,50 with radius 15
# rectangle with a height of 40

car1 = Car(Point(50, 50), 15, Point(100,50), 15, 40)
car1.draw( new_win )

# color the wheels grey with black tires, and the body pink
car1.set_color('black', 'grey', 'pink' )

# make the car move on the screen
car1.animate(new_win, 1, 0, 400)

new_win.mainloop()
```



   The size of the wheel is given only by the radius of the tire circle. You can compute the radius of the wheel circle as a percentage of the radius of the tire circle, e.g. 60%.

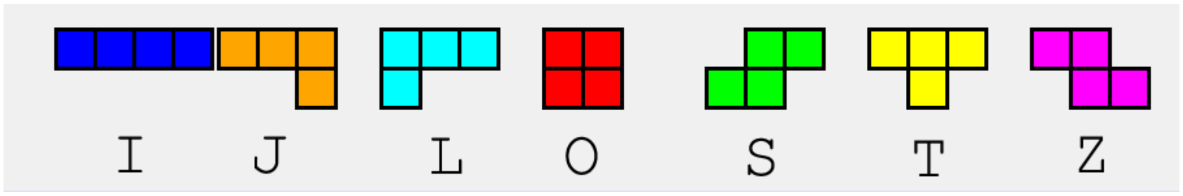
Exercise 4.3 – Tetrominoes

This exercise should be done after you've completed the written exercises. You may want to wait until after Thursday's lecture to start.

Tetris is deemed by some to be the most popular video game of all time. It is a puzzle game developed by Alexey Pajitnov in 1984 while he was working at the Academy of Science of the former USSR in Moscow. There have been hundreds of variants of the game developed since.

We are going to create our own version of the basic Tetris game for the final project. The goal of this exercise is to get familiar with the game and to create the shapes (also called tetrominoes) used in the game. Here are several links where you can get a taste of the game if you've never played it before -
`http://www.tetrisfriends.com/games/Survival/game.php` (fancy gui) and and
`http://vadim.oversigma.com/games/gbt.html` (uses the Green Building as a screen for playing the game). Just remember you need to stop playing at some point `:D`.

Each of the tetrominoes *has* 4 blocks; a block *is* a rectangle (more specifically, a square). Notice some relationships here - 'Has' signifies containment, or an attribute, while 'is' signifies inheritance. Here is an example of the basic tetrominoes:



4.3.1 - Blocks

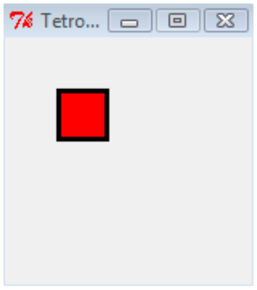
The tetris board is typically 10x20 squares, where a square has a width of 30 pixels. Each block occupies a single square at a time. For this problem, we'll think of the board not in terms of pixels but in terms of squares - so we'll pass in Points in terms of square position, and your methods should convert these into pixels for display purposes.

Create a new `Block` class that inherits from the `Rectangle` class and save it in a file called `tetrominoes.py`. It should have `x` and `y` attributes that correspond to the position of the block on the tetris board, in terms of squares, *not* pixels. The position `(0,0)` is the top left corner of the board, and `(9, 19)` is the bottom right corner.

Here is how you can use the block:

```
win = GraphWin("Tetrominoes", 150, 150)

# the block is drawn at position (1, 1) on the board
block = Block(Point(1, 1), 'red')
# the __init__ method for your block should deal with converting
# the Point into pixels
block.draw(win)

win.mainloop()
```

and what it will look like on a 5x5 board:



Hint: Think about how to initialize the Rectangle superclass: remember that we initialize a superclass by calling its init method, ie, `superclass.__init__(self, param1, param2, ...)`

4.3.2 - Tetromino

Create a `Shape` class that has a *list* of *blocks* as an attribute, and a `draw` method.

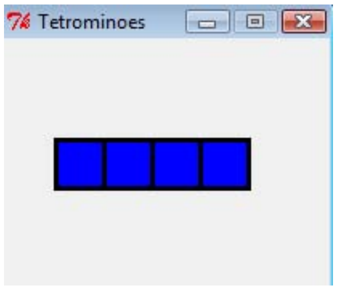
Once you have your `Shape` class made, add in this code for the `I_shape` class, a subclass of the `Shape` class.

```
class I_shape(Shape):
    def __init__(self, center):
        coords = [Point(center.x - 2, center.y),
                  Point(center.x - 1, center.y),
                  Point(center.x    , center.y),
                  Point(center.x + 1, center.y)]
        Shape.__init__(self, coords, "blue")
```

The parameter `center` is a `Point` that holds the position of the central block in the shape.

Verify that your code displays the `I_shape` correctly. Here is an example test:

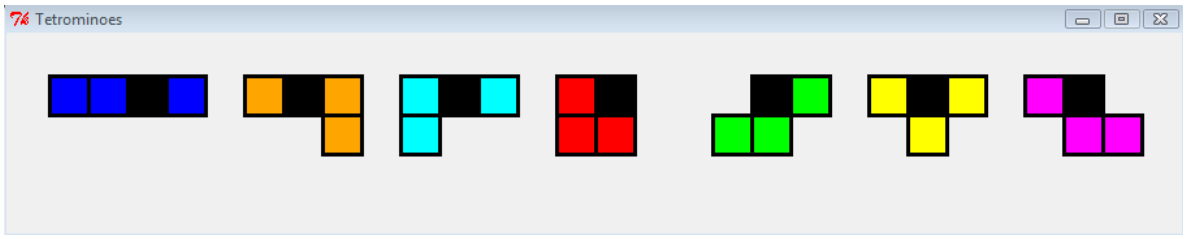
```
win = GraphWin("Tetrominoes", 200, 150)

shape = I_shape(Point(3, 1))
shape.draw(win)

win.mainloop()
```



In the code above, the third block of the I shape is drawn at position (3, 1).

4.3.3 - Many Tetrominoes

Now create subclasses for each of the other 6 shapes. One thing you need to know is, for each shape, which block is the center block. Here are all the shapes with their center block marked in black.



Here is example code for displaying all the shapes above (of course, all your blocks should be the same color- the black blocks are only to represent the 'center' block for each shape):

```
win = GraphWin("Tetrominoes", 900, 150)
# a list of shape classes
tetrominoes = [I_shape, J_shape, L_shape, O_shape, S_shape, T_shape, Z_shape]
x = 3
for tetromino in tetrominoes:
    shape = tetromino(Point(x, 1))
    shape.draw(win)
    x += 4

win.mainloop()
```

Note: you do not have to follow the color scheme of the tetrominoes shown in the picture. Refer to `rgb.txt` for a list of colors Python recognizes.

Exercise OPT.1 – A Digital Clock!

Create a file `dig_clock.py`; do your work in this file and turn in any progress you make!

1. Drawing Text

   The code below is an example of how to draw text on the screen:

   ```
   from graphics import *

   # create the graphics window
   new_win = GraphWin("Digital Clock", 300, 300)

   # create a text objects centered at (100, 100)
   msg1 = Text( Point( 100, 100 ), "Hello, world!" )
   msg1.draw( new_win )

   # process events
   new_win.mainloop()
   ```

   Run your program and make sure the string prints on the screen.

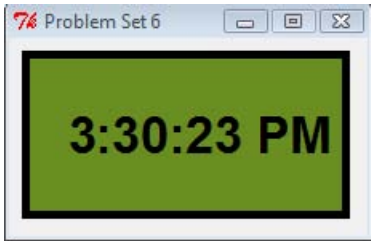
   Try changing the font size and style and the color of the text. Look at the `setSize, setStyle`, and `setTextColor` methods in the documentation. All the set methods that change the attributes of the graphics object, automatically update its appearance on the screen. You can use the list of colors here.

2. Drawing a Digital Clock

   In `dig_clock.py`, create a class called `DigitalClock` that has attributes hour, minute, second and pos, and a draw method. The attributes store the time in military time, i.e. 3:30pm will be hour = 15, minute = 30, second = 23 and the position - the upper left corner of the rectangle face. Here is the code on how to use it:

   ```
   from graphics import *

   # DigitalClock class definition goes here

   new_win = GraphWin("Digital Clock", 300, 300)
   clock = DigitalClock(15, 30, 23)
   clock.draw(new_win)

   new_win.mainloop()
   ```

And an example output:



Feel free to choose the appearance of your clock.

Hint: You should add extra methods to help you draw the clock, e.g. a method for drawing the face, a method for drawing the text, a method returning the time as string. Choose appropriate names for your methods.

3. Updating the clock (Super optional)

   Now you probably created a text object to display the time. Make it an attribute of the clock. Then add an update method that will update the time - both the object attributes and the display on the screen. Think about how you would increment the time. You may want to add other methods to help you. Take a look at the setText function on the Text class.

   Note: The setText method will automatically redraw the text for you. You do not (and should not) call the draw method on the Text object again. You can only draw an object to the screen once.

   You can create a tick method that would call update every second similar to the animate method in the previous exercise.

   Hint: One thing you will have to worry about is handling scenarios, e.g. 05:35:59. The next time the clock updates it should show 05:36:00, not 05:35:60. Similarly for the minutes and hours. The modulus operator is your friend here.

   Another hint: Here is an easy way to avoid trying to handle a lot of different cases. When you update, you first convert the time into seconds from the beginning of the day, then do the update, and then convert back to hour, minute, second. For example,

```
    Current time: 01:01:01 ==> 1*3600 + 1*60 + 1 = 3661
    Update time: 3661 + 1 = 3662
    New time: 3662 ==> 01:01:02
```

   Now you will only need to worry about how to handle updating 23:59:59 to 00:00:00. You may want to add extra methods to help you with this functionality - e.g. a method for converting the time to seconds, a method for splitting it back into hours, minutes, seconds, etc.

Exercise OPT.2 – An Analog Clock!

Wow, I can't believe you've gotten so far! Well we're not going to give you many hints here. Look at the `graphics.pdf` handout for more help on how to draw circles and see if you can draw an analog clock face, and then figure out how to animate the hands. A lot of the code you'll need will be duplicated from the digital clock - in fact, I structured this with a `Clock` superclass that held all the timekeeping functionality, and two subclasses - `AnalogClock` and `DigitalClock` that *inherited* time functionalities from the `Clock` class but implemented different `draw` and `update` methods. Here's how my analog clock looks: