Instructions

These exercises are designed to help cement the concepts from lecture and give you an opportunity to apply them yourself. You are welcome to discuss and share ideas with each other, but should write the answers and code yourself.
Some questions ask you to complete a checkoff, which means you need to come to office hours at this link and talk to a staff member. Staff members may complete checkoffs for multiple students at one time, meaning you may not answer every question yourself but instead should listen to what your peers have to say and ask questions! Collaboration is an important part of computer science.

1 Tasks

1.1 Creating an Object - create.py

We encourage you to discuss the following questions with your peers. Write down your answers, so that you can explain your thinking to a course staff later.

1. Why are classes useful if we already have functions?
2. What is the purpose of the `__init__()` function? When is the function called?

If you are not sure about the answer don't worry. You might have a deeper understanding after finishing the following coding task.

(a) Create a file named **create.py** and write your code in it. First, create a new class called `NewClass`.

(b) Fill in `NewClass` so that it is created using `__init__()`. The parameters for this method are `self` and `the_list`. Save `the_list` to `self.the_list`.

(c) Create an object instance of `NewClass` called `new_object`, giving it a list of ten integers as parameters. Test that it works by printing `new_object.the_list`.

(d) Add a method to `NewClass` called `print_params()` that prints each item in `the_list` on a separate line. Test it before moving on.

(e) Add a method to `NewClass` called `find_average()` that finds and returns the average of the list. The only parameter for this method is `self`. Test it before moving on.

(f) Add a method to `NewClass` called `find_min()` that finds and returns the minimum of the parameters. The only parameter for this method is `self`. Feel free to use the built-in function min() or write your own function. Test it before moving on.

(g) Add a method to `NewClass` called `find_max()` that finds and returns the maximum of the parameters. The only parameter for this method is `self`. Feel free to use the built-in function max() or write your own function. Test it before moving on.

(h) When you're done, you should have a new class, `NewClass`. **UNDER** your class description, you should have the following lines (or something similar) that tests your `NewClass` methods: First, create your `new_object`. Then, add lines to call the various methods (**print_params()**, **find_average()**, **find_min()**, **find_max()**) and print out the output to confirm that your code is working as expected.

(i) Create another object of `NewClass` called `other_object`. Give it a different list and then use all the methods on those too.

Checkoff #1— Why are classes useful as opposed to functions? What does the `__init__()` function do?

1.2 Using an Object as an Attribute - pet.py

Now we will see how classes can be used in more "real-life" applications. Open **pet.py** and read the existing code. First, there is a `Dog` class,which has 3 instance variables:

- `name` (string)
- `weight` (float)
- `breed` (string)

Now you will see there is a `Person` class, which has 3 instance variables:

- `name` (string)
- `dog` (Dog object)
- `generosity` (float)

(a) Fill in the code for the method `feedDog()` in the `Person` class. We will have the person feed the dog based on its generosity – when this method is called, the person's dog should eat the same amount of food as the person's generosity.

For example, if the person has a generosity of 0.5, the person's dog should eat 0.5 pounds of food when `feedDog()` is called. This should modify the dog's weight. That means if the dog previously weighed 13.5 pounds, he should weigh 14.0 pounds after the `feedDog()` method is called.

(b) Create a `Person` object (`my_person`). The `Person` constructor requires a string for the name, a `Dog` object, and the generosity which should be a float between 0 and 1. For the `Dog` object, you will need to create a `Dog` object and directly pass it into the `Person` constructor. (It will look something like `Person(.., Dog, ...))`.

(c) Check the `Dog`'s weight. Then call the `feedDog()` method on `my_person` and check the `Dog`'s weight again. Make sure the weight changed as it was supposed to. Hint: How can you get the `Dog` object? Is it available through the `Person` object?

Notice that you were able to use an object as an instance variable in another class, and that you could also use `Dog`'s methods inside of a different class, `Person`.

Checkoff #2— Which object – Person or Dog – is "within" the other object? If it were the other way around, would you be able to feed the dog by calling the `feedDog()` method on `my_person`?

1.3 Animating a wheel - animate.py

Now we're going to make our own graphics class – a wheel for a car. (It'll just look like a black donut for simplicity's sake.)

What sorts of things should a wheel do? We want it to be able to be drawn (so we can see it) and be able to move, and we want to be able to get its size, center, and color.

Open up **wheel.py** in Spyder. We will explain different sections of the template file below.

```python
class Wheel():
    def __init__(self, center, wheel_radius, tire_radius):
    #Our wheel is constructed with a center point, and a radius
    #for the wheel and the tire

        self.tire_circle = Circle(center, tire_radius)
        # The tire circle attribute is another object, the circle

        self.wheel_circle = Circle(center, wheel_radius)
    def draw(self, win):
    #We want to be able to draw our wheel, so we make a method for that

        self.tire_circle.draw(win)

        #Here, we are using the draw method built-in to Python.self.tire_circle is a
        #Circle object, so we are calling the draw method on that object.

        self.wheel_circle.draw(win)

    def move(self, dx, dy):
    #We also want to be able to move the wheel (it is a wheel after all!)

        self.tire_circle.move(dx, dy)

        #Python has a built-in move method too.
        #We are calling it on the self.tire_circle Circle object here. move takes
        #two parameters - the change in x and the change in y that it should move

        self.wheel_circle.move(dx, dy)

    def set_color(self, wheel_color, tire_color):
    #We also want to be able to set its color.
    #We have to remake all of these methods because we are making a new class,
    #but it is okay to just tell it to do things on objects we already have.

        self.tire_circle.setFill(tire_color)

        self.wheel_circle.setFill(wheel_color)

    def undraw(self):
    #This method will undraw itself -- get rid of the image

        self.tire_circle.undraw()

        self.wheel_circle.undraw()

    def get_size(self):
        return self.tire_circle.getRadius()

    #Circle has a built-in method that gets the radius -- here we are just
    #calling it on the tire_circle object, which is an instance of Circle.

    def get_center(self):
        return self.tire_circle.getCenter() #getcenter is built into Circle too.

win = GraphWin('Wheel', 320, 240)

w = Wheel(Point(100, 100), 50, 70)

#This creates a new object w that is a Wheel.

w.draw(win)

w.set_color('gray', 'black')
```

```
win.getMouse()

win.close()
```

(a) Next, we are going to animate the wheel! Let's add an `animate` method that will move the wheel across the screen. Remember how last PSET we used a loop and a `sleep` function to make an animation? Well, the `graphics` module makes animations even easier by giving us the `GraphWin` method `after`. (The `sleep` function doesn't work well with graphical user interfaces, or GUIs.)

Our animation will use the move method in the `Wheel` class: the `move` method moves the `Wheel` `dx` units in the x-direction and `dy` units in the y-direction. Now write an `animate` method that looks like:

```
from graphics import *
class Wheel(object):
...

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 we want to move the object in the animation (`n`).

The interesting part is the after method on the GraphWin object. The first parameter is the time in milliseconds after which the `GraphWin` object will make a new call to the second parameter, the `animate` method. (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 parameters that will be passed into the new call to the animate method. Note that we decrements `n` by 1 every time we setup a new call to `animate`.

In other words, the `after` method is basically waiting 100 ms before making a recursive call to `animate`! (We didn't discuss recursion in CS before, but you should be able to understand this process by thinking about the nested function calls. The function animate is run again and again until the condition n¿0 is false. Think about what each call does. Does it make sense that this animates the wheel?)

(b) The program now has an updated `Wheel` class. The `Wheel` object that has been created (you can pick the colors of the tire and wheel to be anything you want) will be used to move the wheel across the screen by 1 unit in the x-direction 100 times. Add a function call that moves the wheel by 1 unit in the x-direction 100 times. Remember you need to draw the wheel before you can move it.

(c) How could you make the `Wheel` move faster? Slower? How could you make the wheel move farther across the screen? There are two ways – come up with both! (Answer by commenting in the file.)

2 Optional Challenge[1]:

2.1 Drawing a car - car.py

Now we are going to draw a car! Add your `Wheel` class to the top of **car.py**.

(a) Using the code in **car.py**, we'll begin with a rectangle.

Run your program, and make sure that the rectangle appears on the screen. Try changing the color and width of the outline of the rectangle. Look at the `setOutline` and `setWidth` methods in the documentation.

[1]optional problems are provided for those who have further interests and want to explore more. You are not responsible for those questions, however.

(b) Next, we will create a `Car` class that will use the `Wheel` class from the previous exercise. (You can do this in the same file **car.py**.) 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 of how we will use the `Car` object. You need to figure out how to write the class `Car` based on the way it is used here:

```
# create the window and canvas to draw on win = GraphWin("Car", 700, 300)

# create a car object
# 1st wheel centered at (50, 50) with tire radius 15 # 2nd wheel centered at
(100, 50) with tire radius 15 # rectangle with a height of 40
car1 = Car(wheel1, wheel2, rect)
car1.draw(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(win, 1, 0, 400)

win.getMouse()

win.close()
```

Note that when we define the `Car`, the size of each wheel is given only by the radius of the tire (outer) circle. You can compute the radius of the wheel (inner) circle as a percentage of the radius of the tire circle, e.g. 60

(c) Test your code. When you are confident that your `Car` class works, you're done!

Submitting your PSET

After you've finished your PSET and checkoff: Log into your Canvas account, find the post for Problem Set 6 in Assignments, and submit all of the files that you created or edited. After you turn in your assignment, you're all done!