A L E X M A L A V E

Assignment: Homework #9

Problem 1:
fraction.py & fractiontest.py

fraction.py is intended to be used as a module by the driver program *fractiontest.py*. Within *fraction.py*, a class, **Fraction**, is defined. The constructor takes two arguments, **num** and **denom**, which it uses to create a Fraction (object). The Fraction, when created, is simplified using private method **__simplify**, which I will discuss later.

The method **getNum** returns the numerator of the Fraction, **self.__num**.

The method **getDen** returns the denominator of the Fraction, **self.__denom**.

The method **setNum** sets the numerator of the Fraction to **num**, and simplifies using the **__simplify** method.

The method **setDen** sets the denominator of the Fraction to **denom**, and simplifies using the **__simplify** method.

The method **getReciprocal** returns the reciprocal of the Fraction by creating a Fraction object with the old denominator as the new numerator, and vice versa.

The private method **__simplify** changes the Fraction object given to its simplest form (i.e. 20/100 would be made into 1/5). For the most part, to accomplish this, it determines which of the two, between the absolute values of numerator and denominator, is smallest, and utilizes a while loop to subtract from that number until both the numerator and denominator can be evenly divided by that number. When that happens, the loop breaks, otherwise, the loop goes on and it's determined that the Fraction object is already at its base form. Once that number is found, the actual values of the Fraction object are divided by it in order to transform the Fraction into its base form.

The methods **__str__**, **__float__**, and **__int__** make it possible to convert a Fraction object into a string (concatenates the string of the numerator and string of the denominator), float (divides the numerator by the denominator), or integer (integer divides the numerator by the denominator) value, respectively.

The methods **__add__**, **__sub__**, **__mul__**, and **__truediv__** make it possible to add (by splitting the Fraction, finding a common denominator, and adding across), subtract (by finding a common denominator, and subtracting across), multiply (by multiplying across), and divide (by multiplying across by the reciprocal [which is gotten through the **getReciprocal** method] of the second Fraction) two Fractions, respectively.

The methods **__eq__**, **__ne__**, **__lt__**, **__le__**, **__gt__**, and **__ge__** all convert the two Fraction objects involved into floating point numbers, and then compares the two floats with their respective operands which are: ==, !=, <, <=. >, and >=. They return *True* if the comparison works, and *False* otherwise.

Finally, the file *fractiontest.py* tests the majority of the methods used in *fraction.py*, so as to ensure that they work as intended. The code should be run as follows: *python3 fractiontest.py*

Note: this code is not to be run individually, rather it is meant to be imported into a driver program, such as fractiontest.py.

Problem 2:
ball.py & ballworld.py

ball.py is intended to be used as a module by the driver program *ballworld.py*. Within *ball.py*, a class, **Ball**, is defined. The constructor takes no arguments. Multiple instance variables are defined that determine specific attributes of the ball (position, velocity, size, color, and shape).

The method **setPos** updates the instance variables used to keep track of the ball object's position, and also moves the ball to the given position. This method takes two arguments (x,y).

The method **setColor** updates the instance variables used to create colors and also changes the color of the ball object. This method takes three arguments (r,g,b).

The method **setSize** updates the instance variable used to keep track of the ball object's size, and also changes the size of the ball. This method takes one argument (size).

The method **setVel** updates the instance variables used to keep track of the ball object's velocity. This method takes two arguments (x,y).

The method **update** is used to check if the ball object is within the bounds of the screen, if not, the ball object's velocity is inverted so as simulate bouncing. This method takes no arguments.

The method **move** adds the velocity (in respect to the axes they affect) to the position of the ball. This simulates movement for the ball. This method takes no arguments.

The class **BreathingBall** is a child of **Ball**, and has one additional instance variable, which is used to create a fluctuation in the ball's size so as to simulate a breathing effect. The **Ball** class's update method is also overwritten in order to put this effect in action.

The class **WarpBall** is also a child of **Ball**, and has no additional instance variables. The **Ball** class's update method is overwritten here, too, so that instead of “bouncing” off of the walls, the ball instead “teleports” to the opposite side and maintains its velocity.

Finally, the file ballworld.*py* uses *ball.py* in order to create an environment within a confined region where 50 balls of varying types (normal, breathing, or warping) are generated and put to work. The resulting animation is a rather chaotic, but somewhat cool-looking animation. The code should be run as follows: *python3 ballworld.py*

Note: this code is not to be run individually, rather it is meant to be imported into a driver program, such as ballworld.py.

Extra Credit:

ball.py & ballworld2.py

A new class was added to *ball.py* called **BallWorld**. Its constructor creates an instance variable out of a list of ball objects provided (by *ballworld2.py*, which has also been altered to provide such a list).

The method **checkCollision** checks whether a ball object is within another by using the equation of a circle: *x^2 + y^2 = r^2*. The logic behind it is creating a larger circle with the combined radii of two other circles (the radii of two given ball objects). If the balls are within created circle's diameter (when its radius is the smallest it can be, which is when the radii of the two ball objects are added together without constants), the ball objects must be touching. The problem with my simulation is the fact that some balls overlap, since my hitboxes for the ball objects are off. The reason for this is the fact that the canvas where the balls are and the radii of the balls are not measured in the same units. I don't know the exact conversion, so I use an estimation (I multiply the sum of the radii by 1/35, since that seemed to give the best results in my tests.

The method **bounce** makes the balls “bounce” off of each other in a simple way, much like the normal Ball

“bounces” off of the walls. Since we didn't cover vectors in class too much, I didn't feel the need to do vector math, in addition to being short on the knowledge of how to use the objects in Python (not sure if I would need a module, though I think I would).

Lastly, the method **update** checks each ball's position in relation to that of all the others (in the instance variable made by the constructor of the class **BallWorld**), every step. If the balls are within range of one another (are touching), they bounce off of each other (in a rather physically inaccurate way, due to the lack of vector math, but, still).