ballworld.py

This program consists of a class named **Ball**, which has 2 further subclasses named **BreathingBall** and **WarpBall**. **Ball** consists of a **constructor** and the following method functions: **setPos()**, **setColor()**, **setSize()**, **update()**, and **move()**. **BreathingBall** and **WarpBall** import the constructor from their superclass. **BreathingBall** also imports **update()** from its super class and then overrides it to perform some additional functions. **WarpBall** doesn't import **update** from its superclass and updates with its particular method. The purpose of this program is to make a simulation of balls which are walking in a window in a random fashion.

Ball's constructor here takes no external input and randomly generates all the instance and other variables to be later used in the program. It also sets up a turtle as a circle for given parameters. **setPos()** sets a circle to particular coordinates within a window. **setColor()** sets a circle to a given color. **setSize()** uses sets a circle size to a given radius. **update()** makes sure that a circle's given coordinates don't go out of the window. Then it calls the **setColor()** to change a circle's color. **move()** changes a circle's coordinates for a given velocity and sets the circle to the new coordinates by calling the **setPos()** function.

BreathingBall's constructor, as mentioned before, imports Ball's constructor. It also randomly generates an instance variable called **step** which is later used in its **update()** method. The **update()** method uses a sine wave to help generate a radius for the circles. The **step** variable is used here to alter the radius of these circles.

WarpBall makes balls which when hit one side of the turtle window, appear at exactly the other side without any change in speed or direction of the ball. It does this by using comparison operators and allotting the min x or y coordinate to the balls at the maximum and vice versa.

fraction.py

This program consists of only one class named Fraction and no subclasses. This Fraction class consists of a constructor and the following methods: **getNum()**, **getDen()**, **setNum()**, **setDen()**, **getReciprocal()**, **__simplify()**, **__str__()**, **__float__()**, **__int__()**, **__add__()**, **__sub__()**, **__mul__()**, **__truediv__()**, **__eq__()**, **__ne__()**, **__lt__()**, **__le__()**, **__gt__()**, and **__ge__().** The whole purpose of this class is to help us define a fraction object which we can use to perform mathematical operations and comparison with other fractions of the same type.

At first I want to explain the **__simplify()** method. It makes the numerator a 0 if the denominator is 0. If the numerator is 0 and the denominator is not, then it sets the denominator to 1. Otherwise, it cancels out all the common factors between the 2 fractions. If both the numerator and the denominator are negative, it cancels those negatives as well. If only the denominator is negative then it makes it positive and assigns the numerator a negative sign.

getNum(), **getDen()**, **setNum()**, and **setDen()** are self-explanatory as far as their purpose is concerned. **setNum()** and **setDen()** also call the __simplify() method at the end to make the new resulting fractions a bit cleaner.

__str__(), **__float__()**, and **__int__()** return the fraction as strings, floats, or integers.

__add__(), **__sub__()**, **__mul__()**, **__truediv__()** perform the following mathematical functions between fraction objects: addition, subtraction, multiplication, and division. The results are returned as fraction objects. **__mul__()** multiplies the integers with each other and does the same with the denominators. **__truediv__(),** on the other hand, multiplies one fraction's numerator with the other's denominator and vice versa. **__add__()** and **__sub__()** at first calculate the LCM for the combined new fraction. Then it multiplies each fraction's numerator with the quotient obtained by dividing the LCM with the respective numerator's denominator. Then addition or subtraction is done respectively.

getReciprocal() interchanges numerator's and denominator's positions. They are then returned in a Fraction object.

__eq__(), **__ne__()**, **__lt__()**, **__le__()**, **__gt__()**, and **__ge__()** are comparison operations and they determine the following respectively: Equal to, not equal to, less than, less than or equal to, greater than, and greater than and equal to. A Boolean value of true or false is returned depending on the circumstances. The code also accommodates for when there is a 0 in the denominator.