Lesson 11: Pong

In this lesson, we will continue using classes to make a pong game against the computer. To make this game, we will need a class for the paddle and for the ball.

PART 1: Creating the Ball Class

Let’s start by setting up our tkinter window where our game will be played.

```
from tkinter import *
import time

tk = Tk()
canvas = Canvas(tk, width = 800, height = 600)
canvas.pack()
```

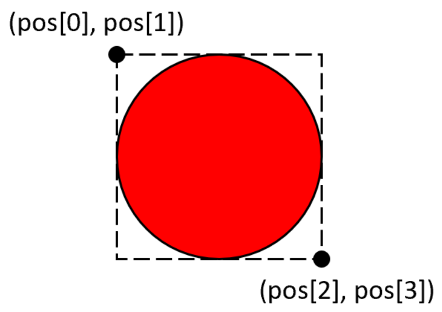
Now, we need to make a ball class. The ball in our game will have a position, size, speed, and a canvas so that it can draw itself. We’ll use the x, y and size parameter to draw our ball. When we draw the ball, we can store the canvas circle thatrepresents our ball as an instance variable called shape. Then, we’ll get the ball’s position using the canvas.coords function which gives you the position of items on the canvas as a tuple of 4 numbers. The first two are the x and y position of the top left corner of our shape. The last two are the x and y position of the bottom right corner of the shape.

```
class Ball:

    def __init__(self, x, y, size, speed, canvas):
        self.shape = canvas.create_oval(x, y, x+size, y+size, fill = "red")
        self.pos = canvas.coords(self.shape)
        self.canvas = canvas
```

The speed is what will control how fast our ball will move. The value of this variable will not change. Our xVel and yVel variables will control the direction. So, at any given time, xVel and yVel can be equal to positive or negative speed.

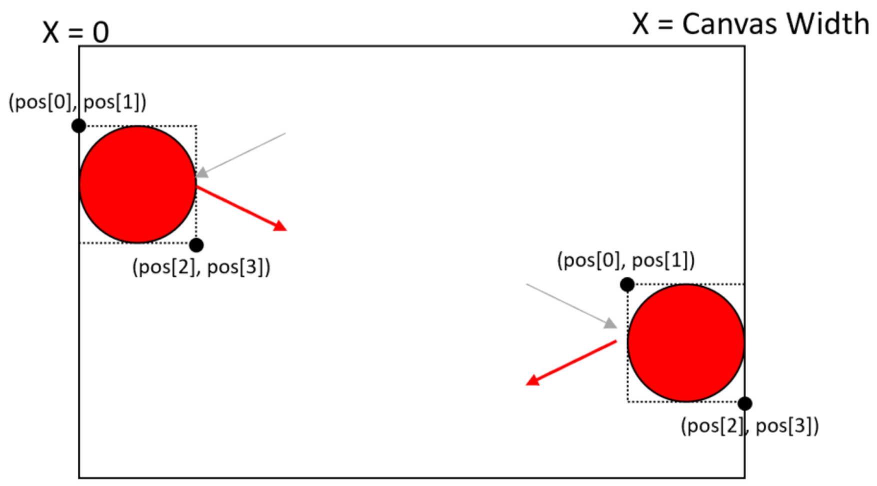
```
class Ball:

    def __init__(self, x, y, size, speed, canvas):
        self.shape = canvas.create_oval(x, y, x+size, y+size, fill = "red")
        self.pos = canvas.coords(self.shape)
        self.canvas = canvas
        self.speed = speed
        self.xVel = speed
        self.yVel = speed
```

We are going to keep our ball from going off the screen. To do that, we'll create a function called containInScreen. First, we need to identify the top, bottom, right, and left edges of our ball. Luckily, our pos variable has all those things already.



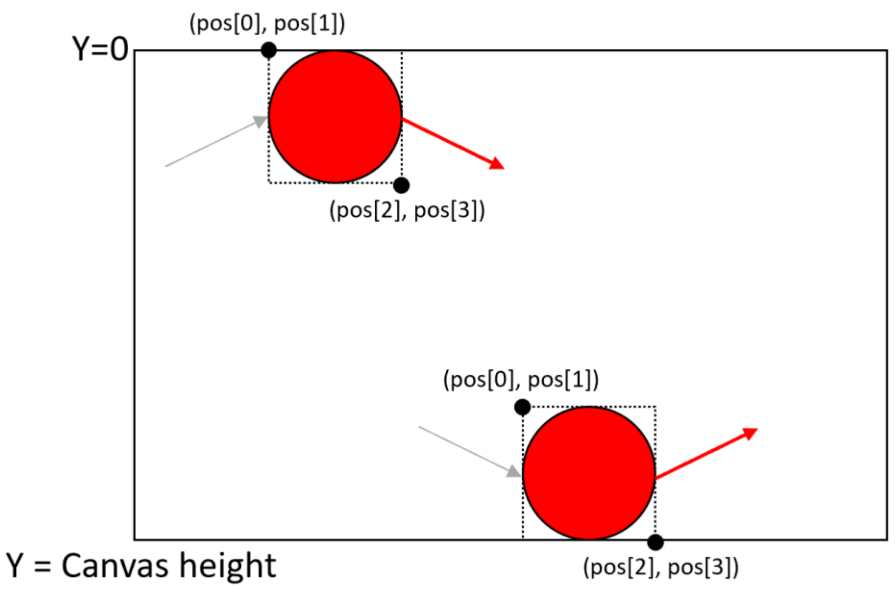
```
class Ball:
    ...

    def containInScreen(self):
        leftX = self.pos[0]
        topY = self.pos[1]
        rightX = self.pos[2]
        bottomY = self.pos[3]
```

"Bouncing" will be done by changing xVel and yVel. If the x-position of the left edge of the ball is less than 0 (where the left edge of the screen is) or the x-position of the right edge of the ball is greater than the width of the canvas (the right edge of the screen), then we should make the ball go in the opposite direction by switching the sign of xVel.



```
    def containInScreen(self):
        leftX = self.pos[0]
        topY = self.pos[1]
        rightX = self.pos[2]
        bottomY = self.pos[3]
        if (leftX <= 0 or rightX >= self.canvas.winfo_width()):
            self.xVel = -1 * self.xVel
```

Similarly, if the top of the ball is less than 0 (the top of the screen) or if the bottom of the ball is greater than the height of the canvas, then we need to switch yVel to the opposite direction.



```
    def containInScreen(self):
        leftX = self.pos[0]
        topY = self.pos[1]
        rightX = self.pos[2]
        bottomY = self.pos[3]
        if (leftX <= 0 or rightX >= self.canvas.winfo_width()):
            self.xVel = -1 * self.xVel
        if (topY <= 0 or _______ >= self.canvas.winfo_height()):
            self.yVel = -1 * self.yVel
```

Next, let's create an update function for our ball. This function will first make sure that the ball is within the bounds of the screen by using the function that we just made. Then it will use the canvas.move function to actually move the ball. Of course, after it moves, we also need to update the ball's position instance variable.

```
class Ball:
    ...

    def update(self):
        self.containInScreen()
        self.canvas.move(self.shape, self.xVel, self.yVel)
        self.pos = self.canvas.coords(self.shape)
```

With the class done, we can now make a ball object. Let's also set up the game loop for our game. In the game loop, we'll update the ball, tk window, and use the time.sleep function to set up the framerate for our game. Also, call tk.update() once before the while loop.

```
class Ball:
    ...

ball = Ball(400, 300, 30, 5, canvas) #(x, y, size, speed, canvas)

tk.update()
while True:
    ball.update()
    tk.update()
    time.sleep(0.01)
```

Test your code and you should see the ball bouncing within the canvas.

PART 2: Making a Moving Paddle

Below your ball class, create the paddle class. In the initialization function, we'll use the x, y, width, and height parameters to draw the shape of our paddle with the canvas. The speed will control how fast the paddle moves. The moveRight and moveLeft Booleans will dictate the direction the paddle should move.

```
class Ball:
    ...

class Paddle:
    def __init__(self, x, y, width, height, speed, canvas):
        self.shape = canvas.create_rectangle(x, y, x+width, y+height,
                                             fill = "green")
        self.pos = canvas.coords(self.shape)
        self.canvas = canvas
        self.speed = speed
        self.moveRight = False
        self.moveLeft = False
```

Let's make a function called setDirection that can change the direction our paddle moves by changing the moveLeft and moveRight instance variables. It will have a direction parameter, so it knows which way to move the paddle.

```
class Paddle:
    ...
    def setDirection(self, direction):
        if (direction == "Left"):
            self.moveLeft = True
            self.moveRight = False
        elif (direction == "Right"):
            self.moveRight = _____
            self.moveLeft = _______
```

Next, we'll make another function that will be used to stop movements in our game.

```
class Paddle:
    ...
    def stopDirection(self, direction):
        if (direction == "Left"):
            self.moveLeft = False
        elif (direction == "Right"):
            self.moveRight = ______
```

The last thing we need to do inside this class is to create an update function that will actually move the paddle. Here, we only allow the paddle to move to the left if the player has pressed the left arrow and if the left edge of the paddle is to the right of the left edge of the screen. Similarly, the paddle will only move to the right if the player has pressed the right arrow and the paddle's right edge is to the left of the right edge of the screen. Last, we update the paddle's position variable with the coords function.

```
class Paddle:
    ...
    def update(self):
        leftX = self.pos[0]
        rightX = self.pos[2]
        if (self.moveLeft and leftX > 0):
            self.canvas.move(self.shape, -self.speed, 0)
        elif (self.moveRight and rightX < self.canvas.winfo_width()):
            self.canvas.move(self.shape, self.speed, 0)
        self.pos = self.canvas.coords(self.shape)
```

Now we can make our paddle. We also need to create two functions outside of the paddle class that we will bind to KeyPress and KeyRelease. These functions will call our paddle's setDirection and stopDirection functions.

```
class Paddle:
    ...
ball = Ball(400, 300, 30, 5, canvas) #(x, y, size, speed, canvas)
paddle = Paddle(500, 540, 150, 10, 5, canvas)

def keyPress(event):
    paddle.setDirection(event.keysym)

def keyRelease(event):
    paddle.stopDirection(event.keysym)

canvas.bind_all("<KeyPress>", keyPress)
canvas.bind_all("<KeyRelease>", keyRelease)
tk.update()
while True:
    ball.update()
    paddle.update()
    ...
```

Test your code. You should now be able to move your paddle left and right without going off the screen.

Part 3: Hitting the Ball

Let's create a bounce function in our ball class. It will have a mode parameter that tells it how the ball should bounce: "Left", "Right", "FastLeft", "FastRight", or "Neutral". All of them will change the yVel to its opposite value, but the rest will change xVel depending on the mode.

```
class Ball:
    ...

    def bounce(self, mode):
        self.yVel = -1 * self.yVel
        if (mode == "Right"):
            self.xVel = self.speed
        elif (mode == "FastRight"):
            self.xVel = 1.25 * self.speed
        elif (mode == "Left"):
            self.xVel = __ * self.speed
        elif (mode == "FastLeft"):
            self.xVel = _____ * self.speed
```

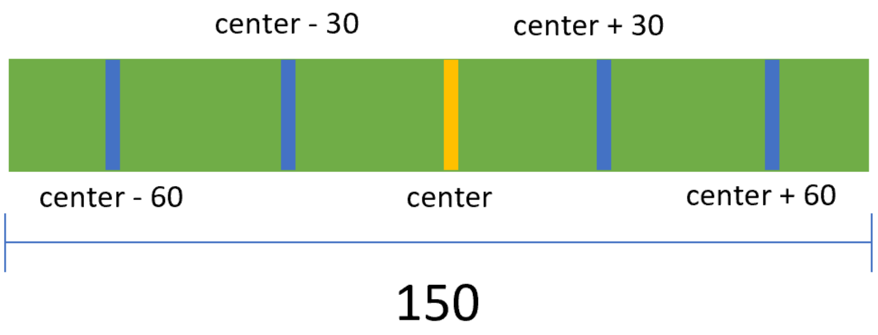
To detect if the paddle has hit a ball, we will create a checkHits function in the paddle class that takes in a ball as an argument. Canvas gives us a useful function that returns a list of canvas shapes that are in a certain area. If we give it the position of our paddle, it will give us a list of items that hit our paddle.

```
class Paddle:

    def checkHits(self, ball):
        hits = self.canvas.find_overlapping(self.pos[0], self.pos[1],
                                            self.pos[2], self.pos[3])
```

Now we can check if ball is in the list to know if the ball is touching the paddle. Once we know that the ball has been hit, we are going to want to know where the center of the ball and the center of the paddle are, so that we can bounce the ball in different ways.

```
class Paddle:

    def checkHits(self, ball):
        hits = self.canvas.find_overlapping(self.pos[0], self.pos[1],
                                            self.pos[2], self.pos[3])
        if (ball.shape in hits):
            ballCenter = (ball.pos[0] + ball.pos[2])/2
            paddleCenter = (self.pos[0] + self.pos[2])/2
```

Here is how our ball will bounce. We'll split the paddle into 5 regions: far left, left, center, right, far right. We'll define these regions using our paddle's center as a reference point and check to see if the ball hits these regions. Then, we will call the ball's bounce function with the appropriate mode.



```
class Paddle:

    def checkHits(self, ball):
        hits = self.canvas.find_overlapping(self.pos[0], self.pos[1],
                                            self.pos[2], self.pos[3])
        if (ball.shape in hits):
            ballCenter = (ball.pos[0] + ball.pos[2])/2
            paddleCenter = (self.pos[0] + self.pos[2])/2
            if (ballCenter < paddleCenter - 60):
                ball.bounce("FastLeft")
            elif (ballCenter < paddleCenter - 30):
                ball.bounce("Left")
            elif (ballCenter < paddleCenter + ___):
                ball.bounce("Neutral")
            elif (ballCenter < paddleCenter + ___):
                ball.bounce("Right")
            else:
                ball.bounce("FastRight")
```

In our while loop, you can call the paddle's checkHits function to hit the ball with your paddle

```
while True:
    ball.update()
    paddle.update()
    paddle.checkHits(ball)
```

Let's test it. Run your code and your ball should now bounce on your paddle.

Before we move on, there is a small bug that we need to fix. It is actually possible for our ball to become "stuck" inside our paddle. To prevent that, we will add an instance variable to the ball class called bounced.

```
class Ball:
    def __init__(self, x, y, size, speed, canvas):
        ...
        self.bounced = False
```

Now, in the ball's bounce function, we will set bounced equal to True.

```
class Ball:
    ...
    def bounce(self, mode):
        self.bounced = True
        ...
```

In the ball's update function, we will set the bounced variable back to False once the ball is no longer touching anything. We can do this by using the find_overlapping function. We just need to check that the list of hits is less than 2 (the ball itself will always be included in the list, so the hit list will always be at least 1).

```
class Ball
    ...
    def update(self):
        self.containInScreen()
        self.canvas.move(self.shape, self.xVel, self.yVel)
        self.pos = self.canvas.coords(self.shape)
        hits = self.canvas.find_overlapping(self.pos[0], self.pos[1],
                                            self.pos[2], self.pos[3])
        if (len(hits) < 2):
            self.bounced = False
```

Now in the paddle class, we need to return right away if the ball has already been bounced. That way the ball will no longer get stuck in the paddle because it will only be bounced once.

```
class Paddle:

    def checkHits(self, ball):
        if (ball.bounced):
            return
        ...
```

Test your code now and make sure you can still hit your ball with your paddle.

Part 4: Making the Computer

First, let's make a paddle for the computer. To do that, we can just make another paddle object.

```
ball = Ball(400, 300, 30, 5, canvas)
paddle = Paddle(500, 540, 150, 10, 5, canvas)
cpuPaddle = Paddle(350, 50, 150, 10, 5, canvas)
```

Now let’s make a function that controls our computer. All the function will do is move the paddle to the left if the ball is to the left of it or move it to right if the ball is to the right of it.

```
def controlComputer():
    ballCenter = (ball.pos[0] + ball.pos[2])/2
    cpuCenter = (cpuPaddle.pos[0] + cpuPaddle.pos[2])/2
    if (ballCenter < cpuCenter):
        cpuPaddle.setDirection("Left")
    elif (ballCenter > __________):
        cpuPaddle._____________(_______)
```

Last, in the while loop, we need to call the same function that we did on the player paddle: update and check hit. We also need to call our controlComputer function to make the computer move its paddle. Let’s also change the while loop so that it ends as soon as the ball hits either the top of the screen or the bottom of the screen.

```
tk.update()
while ball.pos[1] > 0 and ball.pos[3] < canvas.winfo_height():
    ball.update()
    paddle.update()
    paddle.checkHits(ball)
    controlComputer()
    cpuPaddle._______()
    cpuPaddle._________(ball)
    tk.update()
    time.sleep(0.01)
```

Test your code and see if you can beat the computer!

Part 5 – Adding a Replay Option

To make our game replayable, we first need to put our game loop inside a function so that we can call it at any time.

```
def runGame():
    tk.update()
    while ball.pos[1] > 0 and ball.pos[3] < canvas.winfo_height():
        ball.update()
        paddle.update()
        paddle.checkHits(ball)
        controlComputer()
        cpuPaddle._______()
        cpuPaddle.________(ball)
        tk.update()
        time.sleep(0.01)
```

Next, create a startGame function that will reset all our objects back to their default positions. To do that, we are actually going to delete them and create them again. Then we will call runGame at the end to start the game.

```
def startGame(event):
    global ball, paddle, cpuPaddle
    del(ball)
    del(paddle)
    del(cpuPaddle)
    canvas.delete("all")
    ball = Ball(400, 300, 30, 5, canvas)
    paddle = Paddle(500, 540, 150, 10, 5, canvas)
    cpuPaddle = Paddle(350, 50, 150, 10, 5, canvas)
    runGame()
def runGame():
```

Last all we need to do is bind the startGame function to the enter key. Now whenever we want to start the game again, we will just press the enter key. Make sure this line goes outside of any function.

```
def startGame(event):
    ...

canvas.bind_all("<KeyPress-Return>", startGame)

def runGame():
    ...
```

That's it! You should now be able to replay the game after losing or even in the middle of a game.