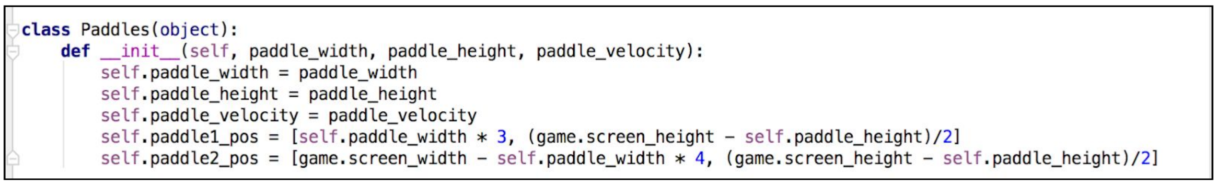
Quick Documentation of Key Principles

Techniques Used:

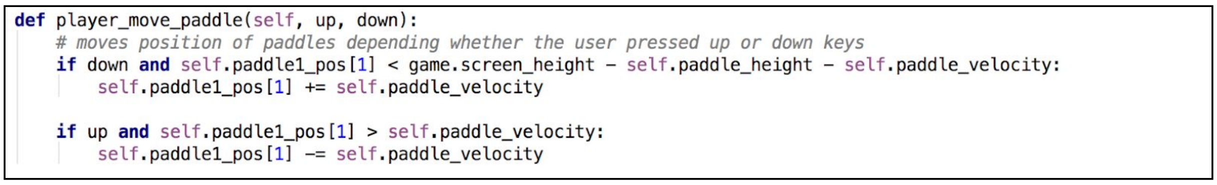
Constructing the paddles



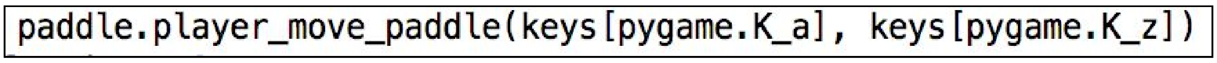
```
class Paddles(object):
    def __init__(self, paddle_width, paddle_height, paddle_velocity):
        self.paddle_width = paddle_width
        self.paddle_height = paddle_height
        self.paddle_velocity = paddle_velocity
        self.paddle1_pos = [self.paddle_width * 3, (game.screen_height - self.paddle_height)/2]
        self.paddle2_pos = [game.screen_width - self.paddle_width * 4, (game.screen_height - self.paddle_height)/2]
```

The most important element here is the paddle position, which is an array storing the [x,y] position of each paddle. Changing the position of the array will move it.

Moving the paddles



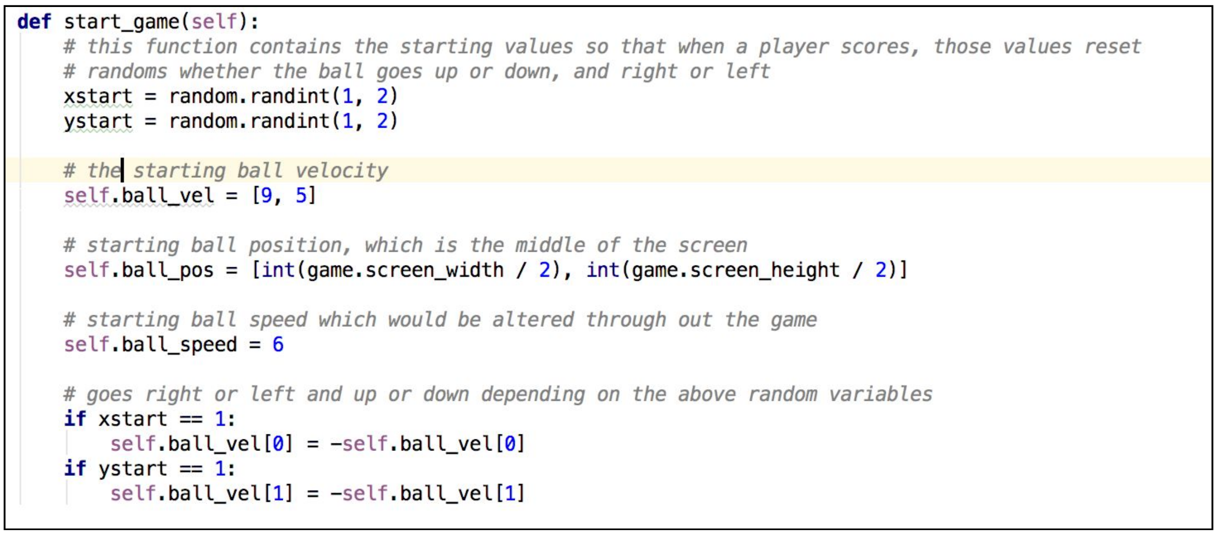
```
def player_move_paddle(self, up, down):
    # moves position of paddles depending whether the user pressed up or down keys
    if down and self.paddle1_pos[1] < game.screen_height - self.paddle_height - self.paddle_velocity:
        self.paddle1_pos[1] += self.paddle_velocity

    if up and self.paddle1_pos[1] > self.paddle_velocity:
        self.paddle1_pos[1] -= self.paddle_velocity
```



```
paddle.player_move_paddle(keys[pygame.K_a], keys[pygame.K_z])
```

The function checks whether the 'up/down' key is pressed and the y position of the paddle is within the boundaries of the screen. If so, it adds or subtracts 'velocity' to the y position, causing the paddle to move up or down.

Starting the game

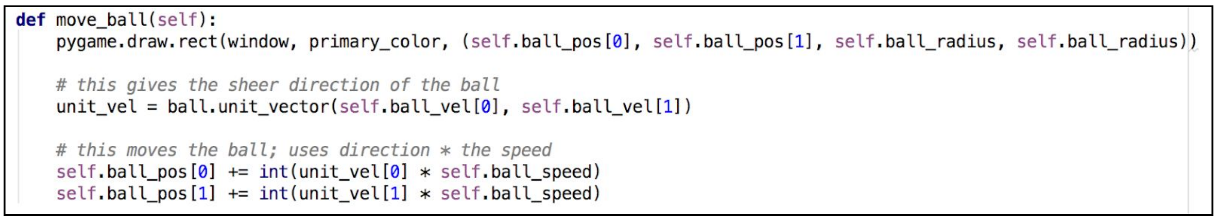


```
def start_game(self):
    # this function contains the starting values so that when a player scores, those values reset
    # randoms whether the ball goes up or down, and right or left
    xstart = random.randint(1, 2)
    ystart = random.randint(1, 2)

    # the starting ball velocity
    self.ball_vel = [9, 5]

    # starting ball position, which is the middle of the screen
    self.ball_pos = [int(game.screen_width / 2), int(game.screen_height / 2)]

    # starting ball speed which would be altered through out the game
    self.ball_speed = 6

    # goes right or left and up or down depending on the above random variables
    if xstart == 1:
        self.ball_vel[0] = -self.ball_vel[0]
    if ystart == 1:
        self.ball_vel[1] = -self.ball_vel[1]
```

This function randomly decides whether the ball at the start of the game will go up or down, and right or left. Also stores basic starting values that need to be reset when player scores.

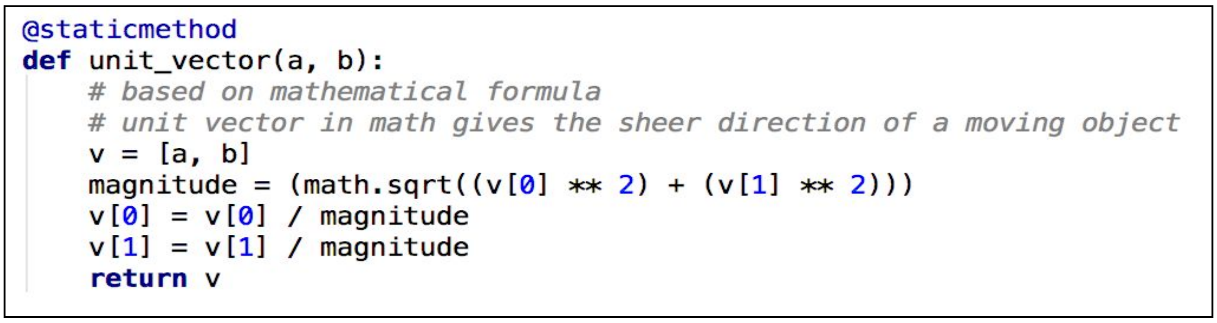
Although the velocity is constant throughout the game, having it in the code means that in the future, the function of changing angles can be added.

Moving the ball



```python
def move_ball(self):
    pygame.draw.rect(window, primary_color, (self.ball_pos[0], self.ball_pos[1], self.ball_radius, self.ball_radius))

    # this gives the sheer direction of the ball
    unit_vel = ball.unit_vector(self.ball_vel[0], self.ball_vel[1])

    # this moves the ball; uses direction * the speed
    self.ball_pos[0] += int(unit_vel[0] * self.ball_speed)
    self.ball_pos[1] += int(unit_vel[1] * self.ball_speed)
```

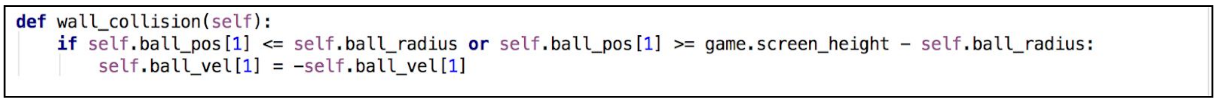
At first, the unit velocity is determined to give the direction of the ball. That unit velocity is then multiplied by the ball speed and added to the ball position, thus moving the ball. Keeping the two seperate allows the future code to manipulate speed without changing the direction.



```python
@staticmethod
def unit_vector(a, b):
    # based on mathematical formula
    # unit vector in math gives the sheer direction of a moving object
    v = [a, b]
    magnitude = (math.sqrt((v[0] ** 2) + (v[1] ** 2)))
    v[0] = v[0] / magnitude
    v[1] = v[1] / magnitude
    return v
```

The unit velocity is determined by dividing each [x, y] by their collective magnitude.

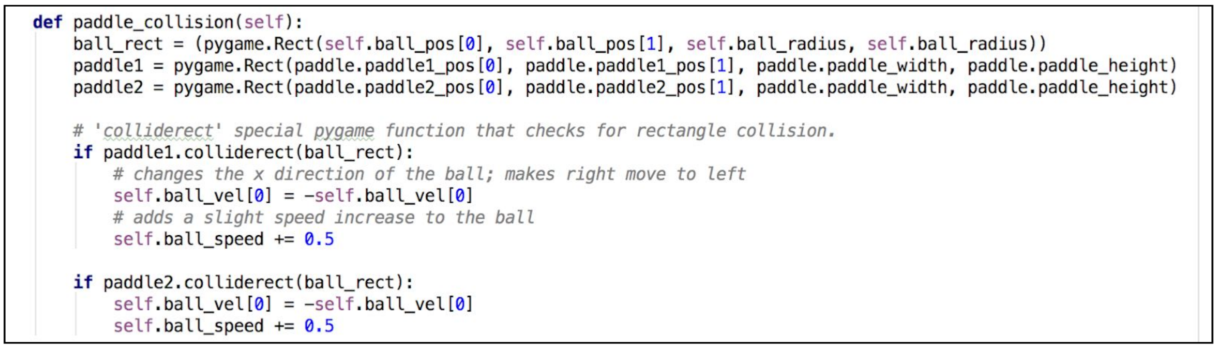
Wall Collision



```python
def wall_collision(self):
    if self.ball_pos[1] <= self.ball_radius or self.ball_pos[1] >= game.screen_height - self.ball_radius:
        self.ball_vel[1] = -self.ball_vel[1]
```

Checks whether the ball is within edges of the screen, and if it is, reverses the ‘y’ value, causing the ball to go opposite direction.

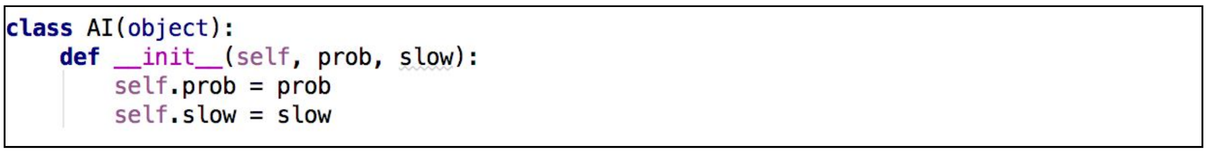
Paddle Collision



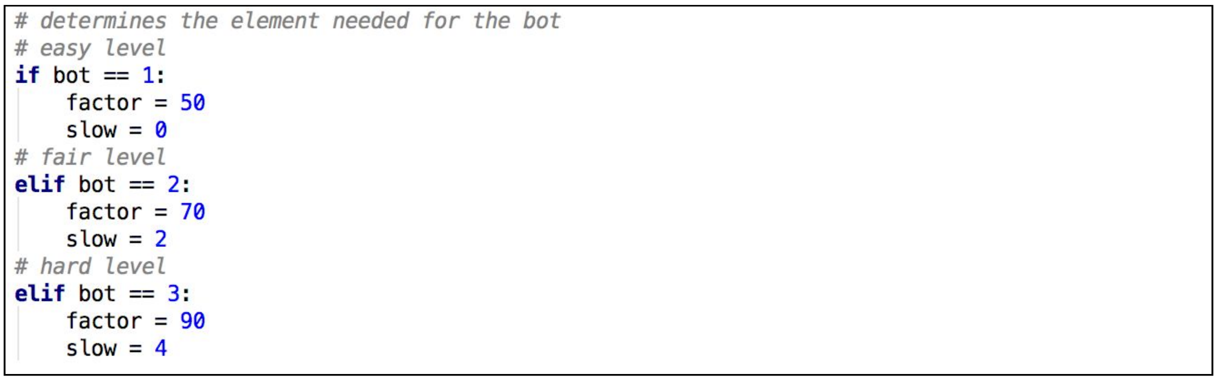
```python
def paddle_collision(self):
    ball_rect = (pygame.Rect(self.ball_pos[0], self.ball_pos[1], self.ball_radius, self.ball_radius))
    paddle1 = pygame.Rect(paddle.paddle1_pos[0], paddle.paddle1_pos[1], paddle.paddle_width, paddle.paddle_height)
    paddle2 = pygame.Rect(paddle.paddle2_pos[0], paddle.paddle2_pos[1], paddle.paddle_width, paddle.paddle_height)

    # 'colliderect' special pygame function that checks for rectangle collision.
    if paddle1.colliderect(ball_rect):
        # changes the x direction of the ball; makes right move to left
        self.ball_vel[0] = -self.ball_vel[0]
        # adds a slight speed increase to the ball
        self.ball_speed += 0.5

    if paddle2.colliderect(ball_rect):
        self.ball_vel[0] = -self.ball_vel[0]
        self.ball_speed += 0.5
```

At first, I used numerical boundaries for collision. However, the pygame function 'colliderect' was more smooth and accurate to the pixel.

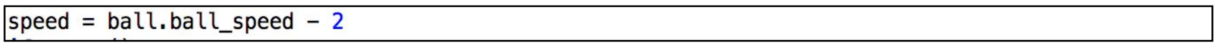
Defining the BOT



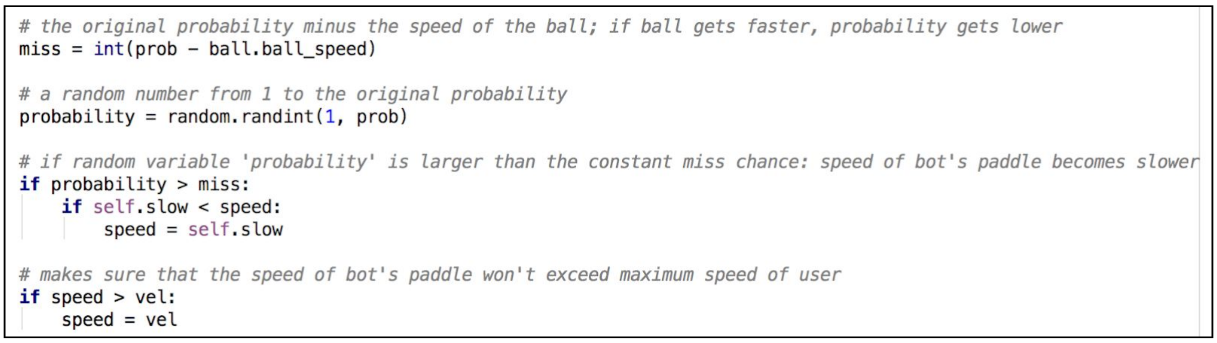
```
class AI(object):
    def __init__(self, prob, slow):
        self.prob = prob
        self.slow = slow
```



```
# determines the element needed for the bot
# easy level
if bot == 1:
    factor = 50
    slow = 0
# fair level
elif bot == 2:
    factor = 70
    slow = 2
# hard level
elif bot == 3:
    factor = 90
    slow = 4
```



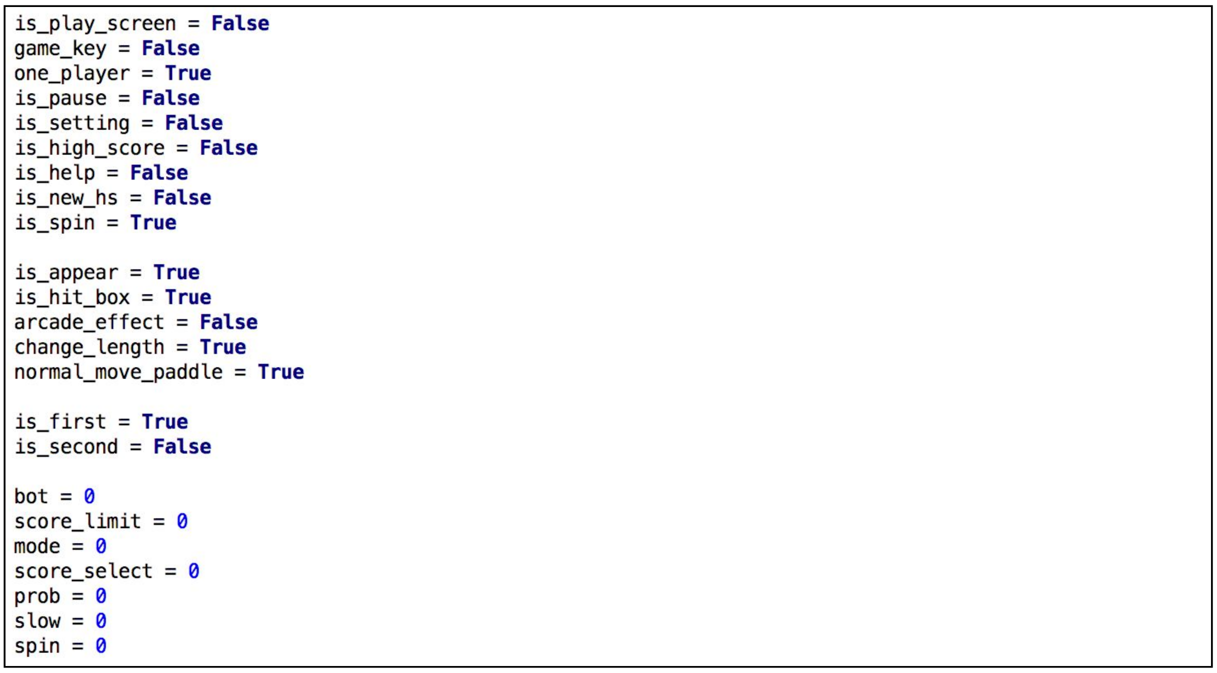
```
speed = ball.ball_speed - 2
```



```
# the original probability minus the speed of the ball; if ball gets faster, probability gets lower
miss = int(prob - ball.ball_speed)

# a random number from 1 to the original probability
probability = random.randint(1, prob)

# if random variable 'probability' is larger than the constant miss chance: speed of bot's paddle becomes slower
if probability > miss:
    if self.slow < speed:
        speed = self.slow

# makes sure that the speed of bot's paddle won't exceed maximum speed of user
if speed > vel:
    speed = vel
```

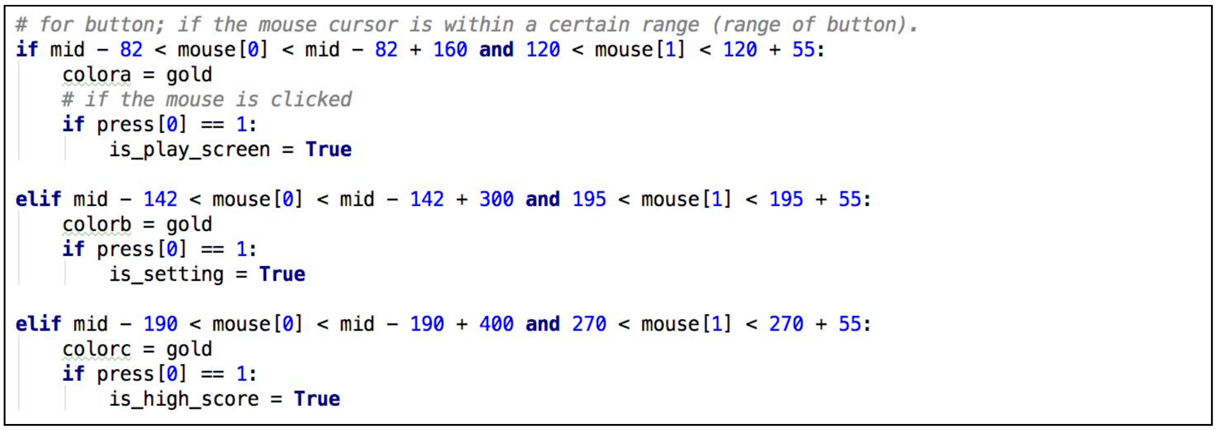
The BOT paddle is travelling at a speed slightly less than the ball speed. Difficulty of the BOT is incorporated by changing the speed of the paddle from 'speed' to 'slow', meaning the paddle will be unable to react to the incoming ball. The 'slow' speed varies from difficulty. Another aspect is the probability of the 'slow' triggering. There is the randomized 'probability (a random number till 'prob')' and there is the 'miss', which is the 'prob' minus the ball speed. This means that with increased ball speed, the chances of the 'slow' triggering is increased, thus decreasing BOT difficulty with a faster ball (just like humans).

Registering Selections



```
is_play_screen = False
game_key = False
one_player = True
is_pause = False
is_setting = False
is_high_score = False
is_help = False
is_new_hs = False
is_spin = True

is_appear = True
is_hit_box = True
arcade_effect = False
change_length = True
normal_move_paddle = True

is_first = True
is_second = False

bot = 0
score_limit = 0
mode = 0
score_select = 0
prob = 0
slow = 0
spin = 0
```

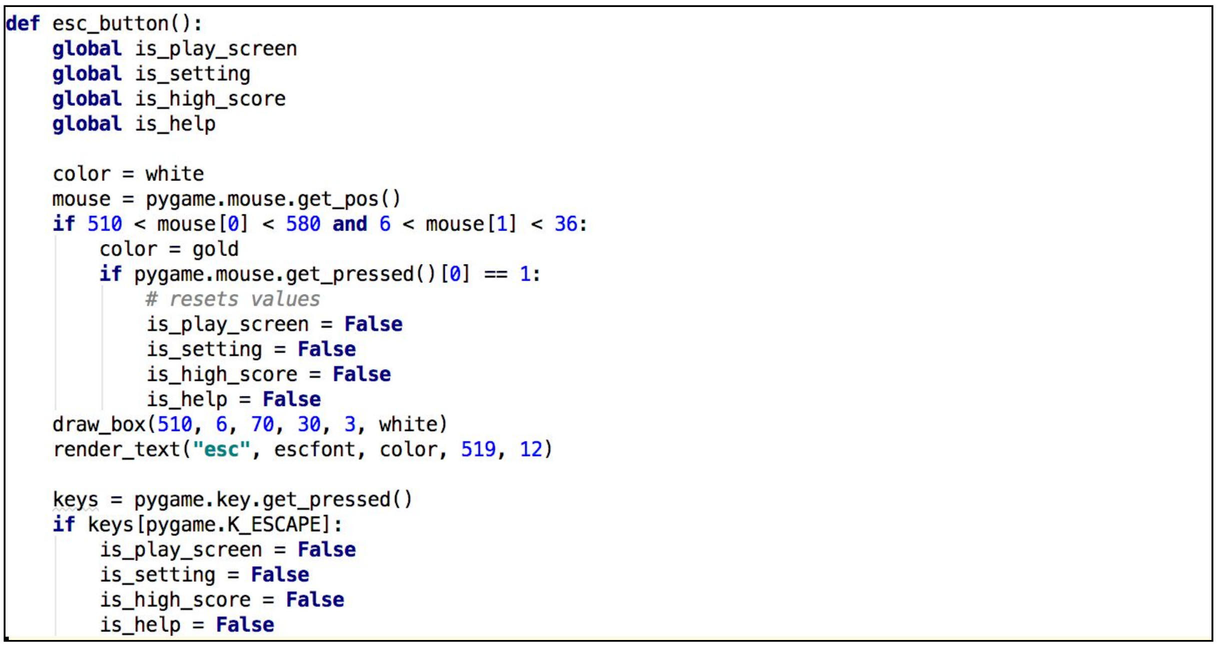
If a user did select a mode or number of players, the algorithm should change the game based on those selections. Thus, it was crucial that the program saved those selections. With that, I used boolean logic to decide whether certain selections have been chosen.



```
# for button; if the mouse cursor is within a certain range (range of button).
if mid - 82 < mouse[0] < mid - 82 + 160 and 120 < mouse[1] < 120 + 55:
    colora = gold
    # if the mouse is clicked
    if press[0] == 1:
        is_play_screen = True

elif mid - 142 < mouse[0] < mid - 142 + 300 and 195 < mouse[1] < 195 + 55:
    colorb = gold
    if press[0] == 1:
        is_setting = True

elif mid - 190 < mouse[0] < mid - 190 + 400 and 270 < mouse[1] < 270 + 55:
    colorc = gold
    if press[0] == 1:
        is_high_score = True
```

Registering selections should work whether the user clicked the button or not. With that, the algorithm checks whether the mouse was with a certain range and then clicked.

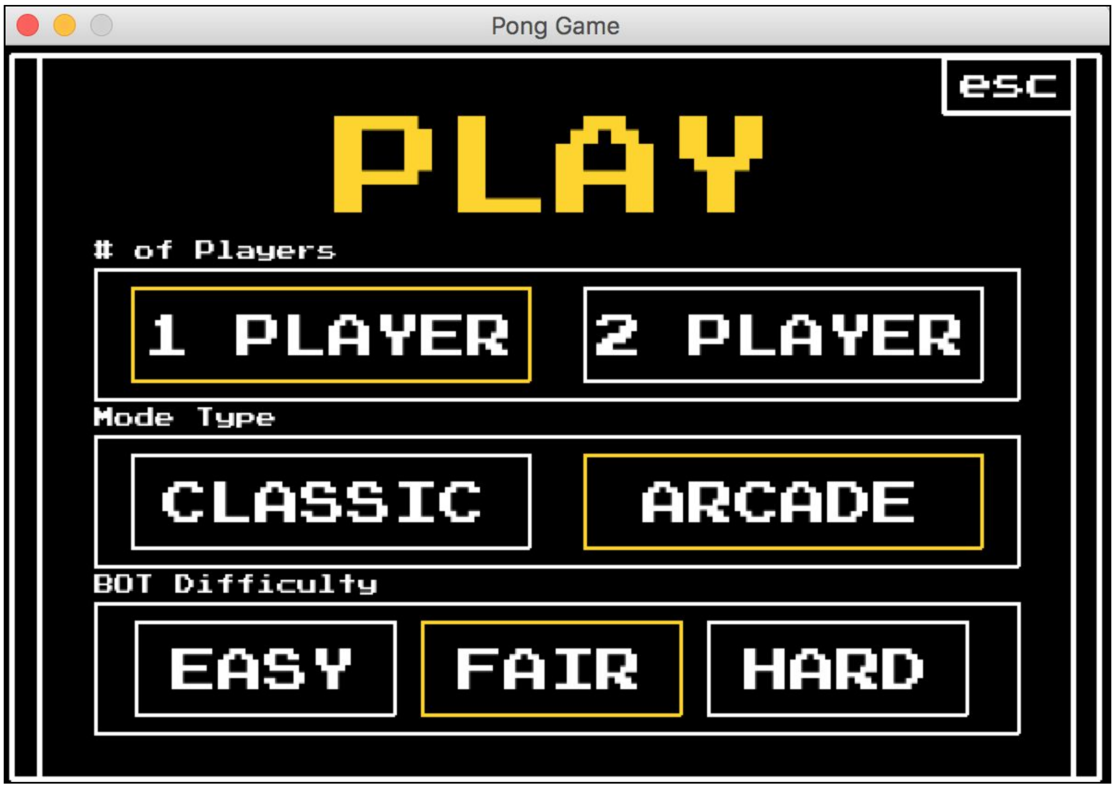
The ‘esc’ Button

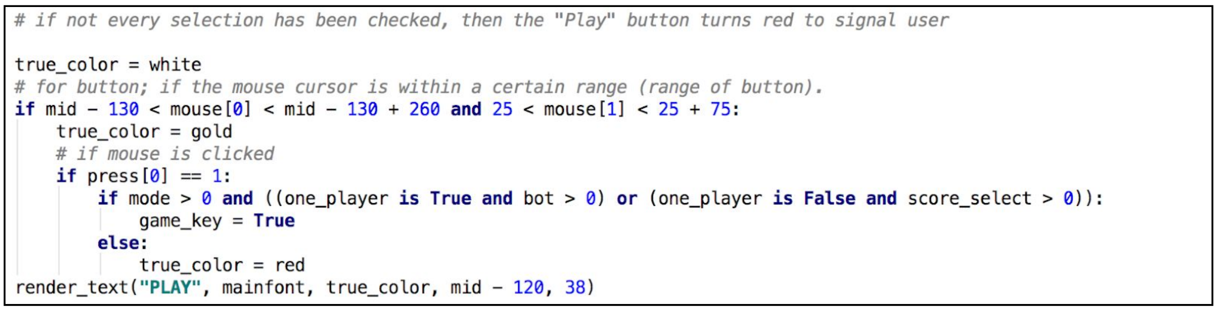


```
def esc_button():
    global is_play_screen
    global is_setting
    global is_high_score
    global is_help

    color = white
    mouse = pygame.mouse.get_pos()
    if 510 < mouse[0] < 580 and 6 < mouse[1] < 36:
        color = gold
        if pygame.mouse.get_pressed()[0] == 1:
            # resets values
            is_play_screen = False
            is_setting = False
            is_high_score = False
            is_help = False
    draw_box(510, 6, 70, 30, 3, white)
    render_text("esc", escfont, color, 519, 12)

    keys = pygame.key.get_pressed()
    if keys[pygame.K_ESCAPE]:
        is_play_screen = False
        is_setting = False
        is_high_score = False
        is_help = False
```

The ‘esc’ button is an example of the importance of boolean logic in my code. To be able to return to the ‘main menu’, the algorithm resets all the necessary variables that come later in my main loop.

Allowing User to Play



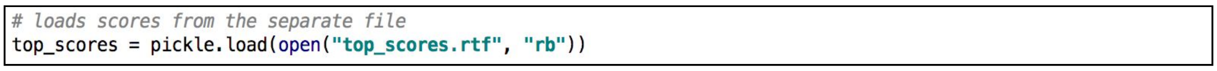


```
# if not every selection has been checked, then the "Play" button turns red to signal user

true_color = white
# for button; if the mouse cursor is within a certain range (range of button).
if mid - 130 < mouse[0] < mid - 130 + 260 and 25 < mouse[1] < 25 + 75:
    true_color = gold
    # if mouse is clicked
    if press[0] == 1:
        if mode > 0 and ((one_player is True and bot > 0) or (one_player is False and score_select > 0)):
            game_key = True
        else:
            true_color = red
render_text("PLAY", mainfont, true_color, mid - 120, 38)
```

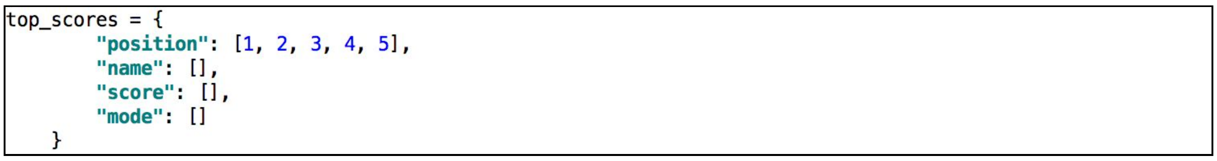
The user must select all three criteria in order to start the game. If not, the algorithm should warn him/her. I used boolean logic to see whether all three criteria have been selected.

Another aspect seen is color change. We see that ‘true_color’ is originally white. But if it is within a certain range, ‘true_color’ changes to gold. But if all three criteria haven’t been selected, it changes the color to red, signaling the user. Based on shrewd placement, color change has been achieved.

Highscore File



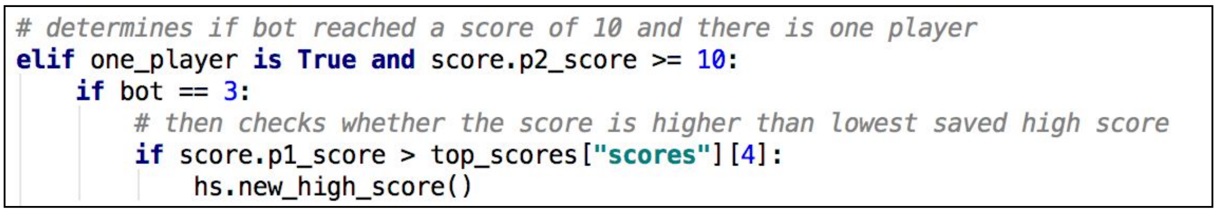
```
# loads scores from the separate file
top_scores = pickle.load(open("top_scores.rtf", "rb"))
```



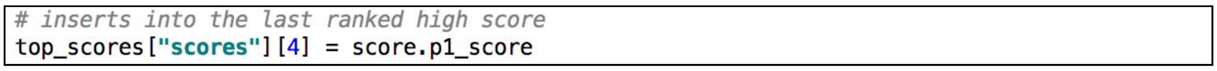
```
top_scores = {
        "position": [1, 2, 3, 4, 5],
        "name": [],
        "score": [],
        "mode": []
    }
```

This is the way the highscores are loaded in a separate file ‘top_scores.rtf’.

Determining Highscores



```
# determines if bot reached a score of 10 and there is one player
elif one_player is True and score.p2_score >= 10:
    if bot == 3:
        # then checks whether the score is higher than lowest saved high score
        if score.p1_score > top_scores["scores"][4]:
            hs.new_high_score()
```



```
# inserts into the last ranked high score
top_scores["scores"][4] = score.p1_score
```

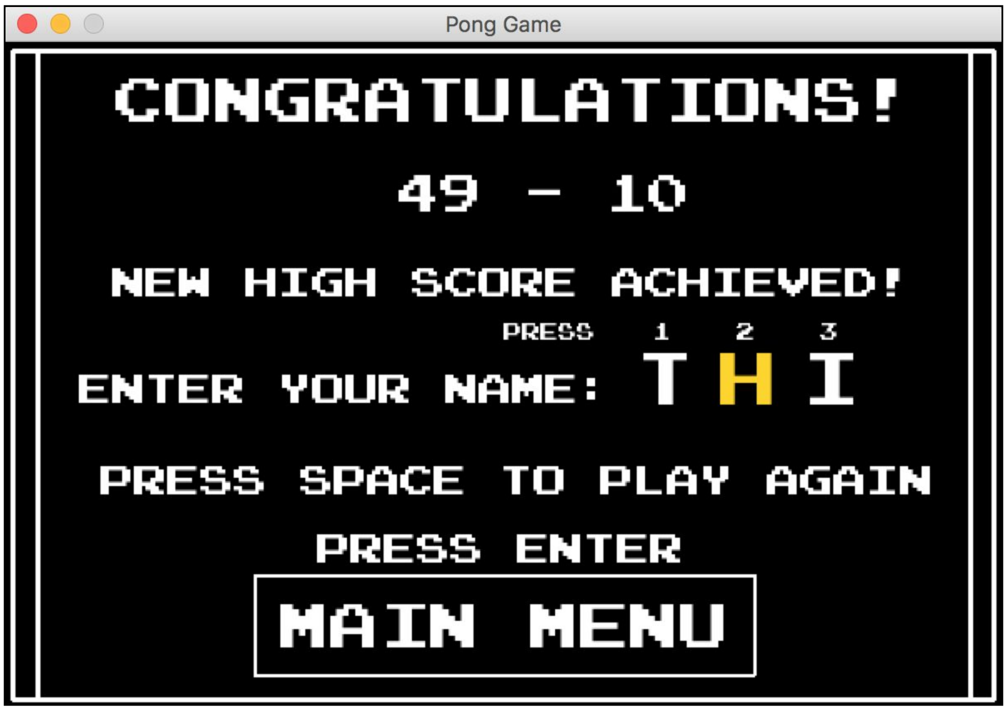
The algorithm first checks whether BOT difficulty is “Hard” and if the score achieved by the user is greater than the lowest saved score. If so, the player score becomes the new lowest saved score, setting up the stage for sorting.

Sorting Highscores

```python
@staticmethod
def sort():
    global top_scores
    for x in range(0, 4):
        minIndex = x
        for j in range(x + 1, 5):
            if top_scores["scores"][j] > top_scores["scores"][minIndex]:
                minIndex = j
        if minIndex != x:
            top_scores["scores"][x], top_scores["scores"][minIndex] = top_scores["scores"][minIndex],top_scores["scores"][x]
            top_scores["names"][x], top_scores["names"][minIndex] = top_scores["names"][minIndex], top_scores["names"][x]
            top_scores["mode"][x], top_scores["mode"][minIndex] = top_scores["mode"][minIndex], top_scores["mode"][x]
```

After the player score becomes the lowest score, it does not mean that it is the lowest in the list; the list then has to be sorted. I decided to use selection sort since my lists consist of only 5 elements, meaning selection sort will result in less procedures. Using nested loops, I was able to compare to elements and make them swap places if necessary.

Entering Highscore Name



Pong Game

CONGRATULATIONS!

49 - 10

NEW HIGH SCORE ACHIEVED!

PRESS 1 2 3

ENTER YOUR NAME: T H I

PRESS SPACE TO PLAY AGAIN

PRESS ENTER

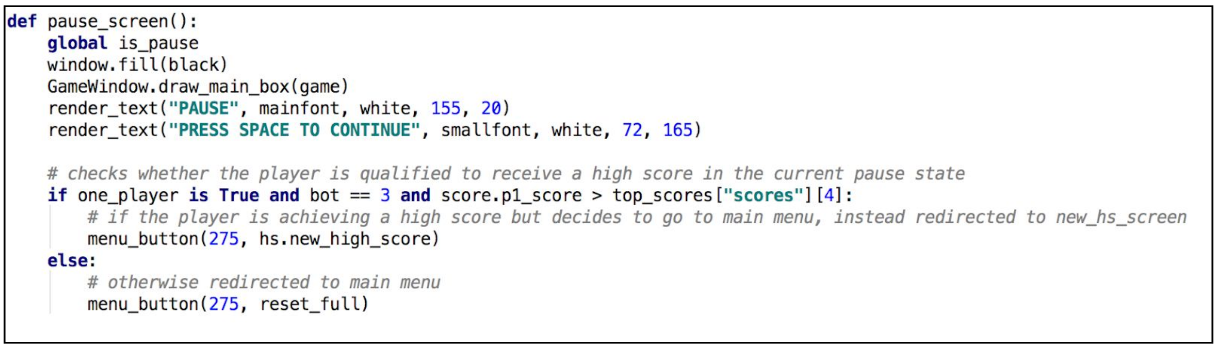
MAIN MENU

```python
keys = pygame.key.get_pressed()
if is_first is True:
    if keys[pygame.K_UP]:
        # delay needed so that if a user presses key button the letters will change one by one.
        pygame.time.delay(150)
        if self.first < 90:
            # originally stores letters in numbers for latter conversion
            self.first += 1
    elif keys[pygame.K_DOWN]:
        pygame.time.delay(150)
        if self.first > 65:
            self.first -= 1
```

```python
# 'chr' converts the numbers into letters
render_text(chr(self.first), subfont, self.color1, 380, 187)
render_text(chr(self.second), subfont, self.color2, 430, 187)
render_text(chr(self.third), subfont, self.color3, 480, 187)
```

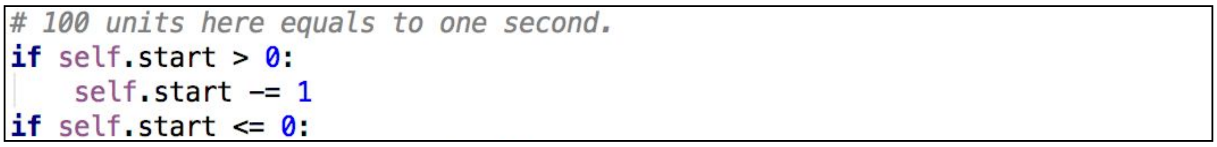
For the user to enter his name when achieving a highscore, I decided to use ASCII code. The user is able to move the ‘up’ and ‘down’ key, which decreases or increases a number which is later converted into alphabet using ASCII.

I also decided to save a user highscore even if he/she returns to main menu.

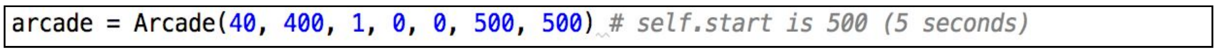


```
def pause_screen():
    global is_pause
    window.fill(black)
    GameWindow.draw_main_box(game)
    render_text("PAUSE", mainfont, white, 155, 20)
    render_text("PRESS SPACE TO CONTINUE", smallfont, white, 72, 165)

    # checks whether the player is qualified to receive a high score in the current pause state
    if one_player is True and bot == 3 and score.p1_score > top_scores["scores"][4]:
        # if the player is achieving a high score but decides to go to main menu, instead redirected to new_hs_screen
        menu_button(275, hs.new_high_score)
    else:
        # otherwise redirected to main menu
        menu_button(275, reset_full)
```

Timing in Arcade

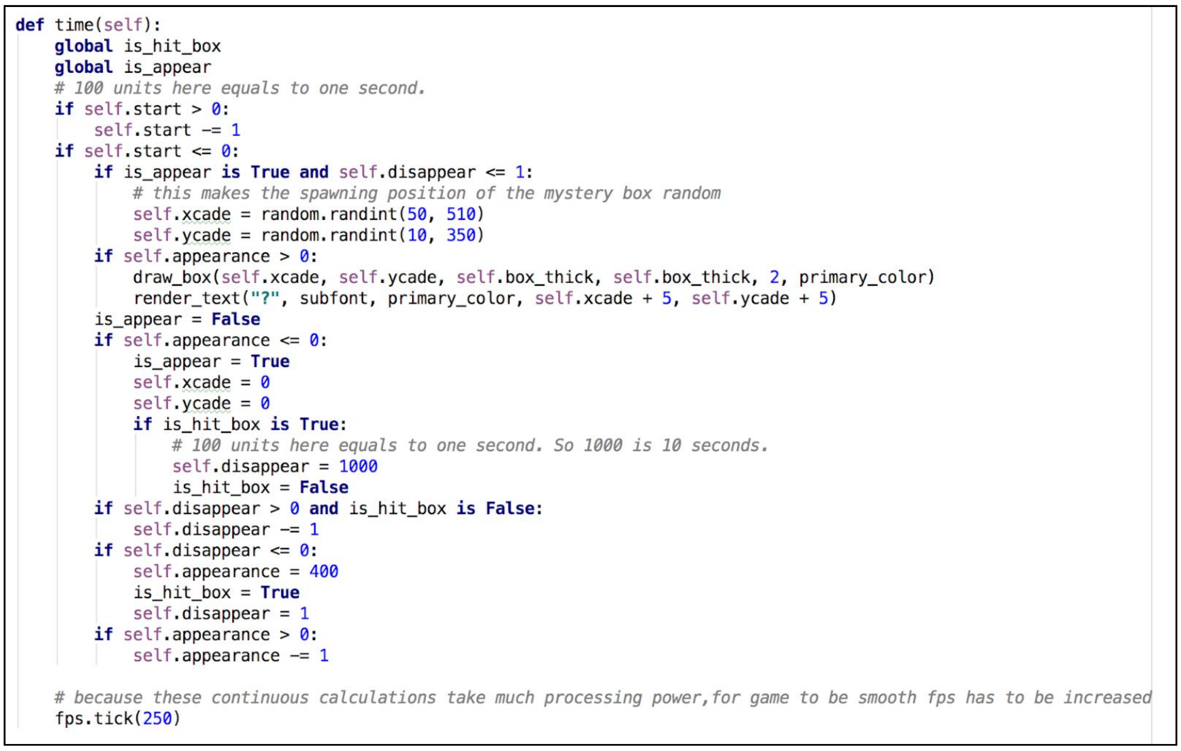


```
# 100 units here equals to one second.
if self.start > 0:
    self.start -= 1
if self.start <= 0:
```



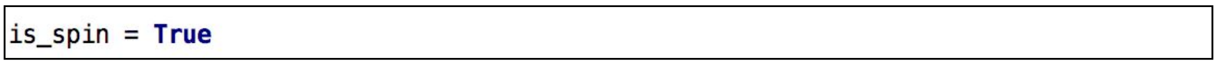
```
arcade = Arcade(40, 400, 1, 0, 0, 500, 500) # self.start is 500 (5 seconds)
```

At the beginning of an arcade game, there should be a 5 second pause before a mystery box spawns. Since my game is inside a continuous loop, ‘-1’ will be constantly subtracted from ‘self.start’, at a speed of -100 for each second. Thus, if I want the pause to last 5 seconds, I have to make the variable be 500.
In all this adds up to a complex series of timings and boolean:

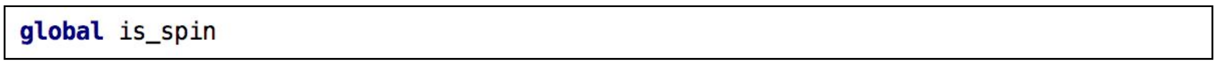


```
def time(self):
    global is_hit_box
    global is_appear
    # 100 units here equals to one second.
    if self.start > 0:
        self.start -= 1
    if self.start <= 0:
        if is_appear is True and self.disappear <= 1:
            # this makes the spawning position of the mystery box random
            self.xcade = random.randint(50, 510)
            self.ycade = random.randint(10, 350)
        if self.appearance > 0:
            draw_box(self.xcade, self.ycade, self.box_thick, self.box_thick, 2, primary_color)
            render_text("?", subfont, primary_color, self.xcade + 5, self.ycade + 5)
        is_appear = False
        if self.appearance <= 0:
            is_appear = True
            self.xcade = 0
            self.ycade = 0
            if is_hit_box is True:
                # 100 units here equals to one second. So 1000 is 10 seconds.
                self.disappear = 1000
                is_hit_box = False
        if self.disappear > 0 and is_hit_box is False:
            self.disappear -= 1
        if self.disappear <= 0:
            self.appearance = 400
            is_hit_box = True
            self.disappear = 1
        if self.appearance > 0:
            self.appearance -= 1

    # because these continuous calculations take much processing power,for game to be smooth fps has to be increased
    fps.tick(250)
```

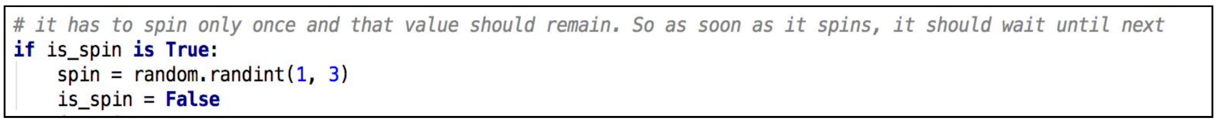
Random in Arcade



```
is_spin = True
```



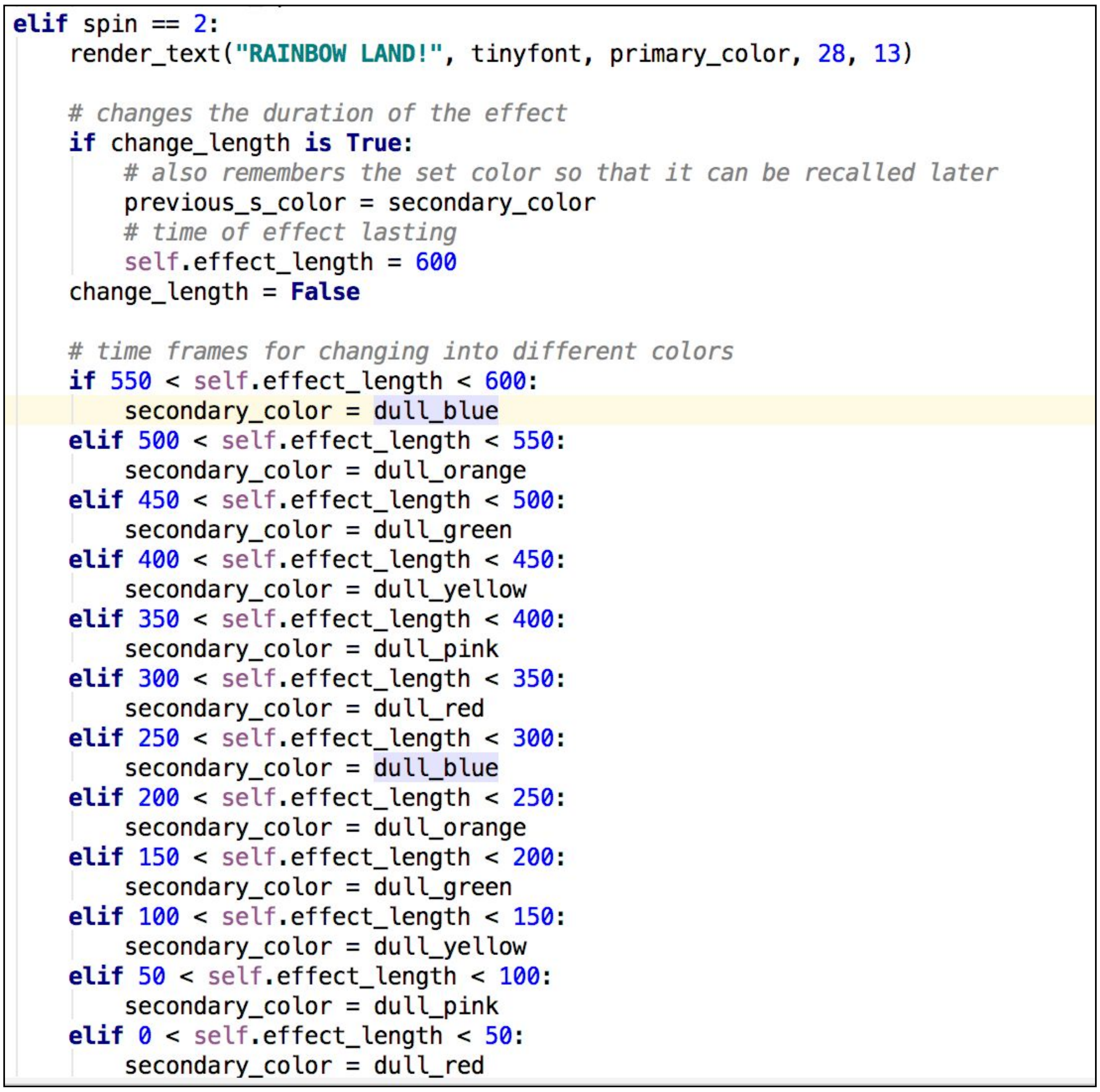
```
global is_spin
```



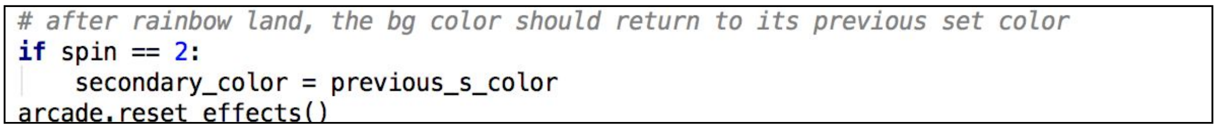
```
# it has to spin only once and that value should remain. So as soon as it spins, it should wait until next
if is_spin is True:
    spin = random.randint(1, 3)
    is_spin = False
```

In many cases, I want a random variable only once every so often. But since my game is in a continuous loop, it would constantly 'spin' over and over. So, I store the variable separate of my function, and then as soon as it is randomed, I lock it in using boolean logic.

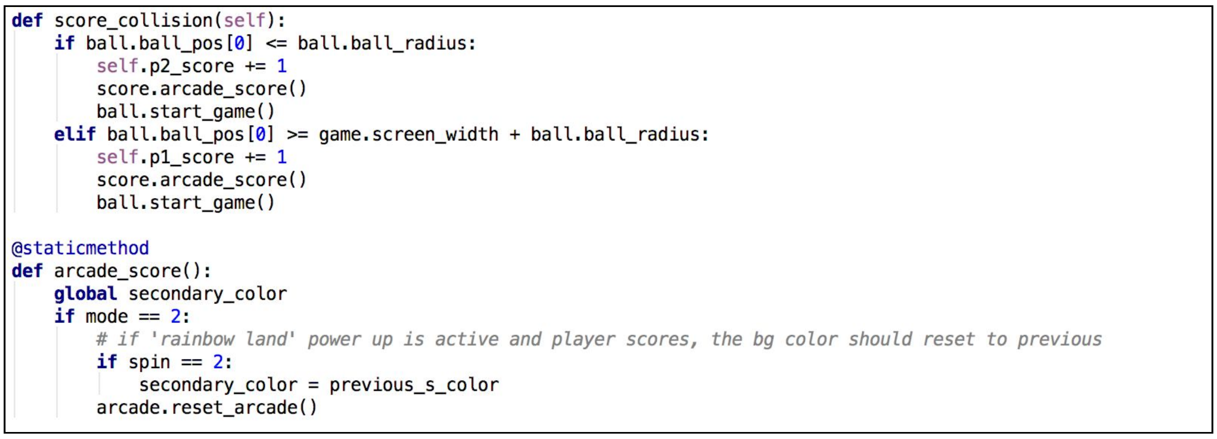
Rainbow Land Power-up!



```
elif spin == 2:
    render_text("RAINBOW LAND!", tinyfont, primary_color, 28, 13)

    # changes the duration of the effect
    if change_length is True:
        # also remembers the set color so that it can be recalled later
        previous_s_color = secondary_color
        # time of effect lasting
        self.effect_length = 600
    change_length = False

    # time frames for changing into different colors
    if 550 < self.effect_length < 600:
        secondary_color = dull_blue
    elif 500 < self.effect_length < 550:
        secondary_color = dull_orange
    elif 450 < self.effect_length < 500:
        secondary_color = dull_green
    elif 400 < self.effect_length < 450:
        secondary_color = dull_yellow
    elif 350 < self.effect_length < 400:
        secondary_color = dull_pink
    elif 300 < self.effect_length < 350:
        secondary_color = dull_red
    elif 250 < self.effect_length < 300:
        secondary_color = dull_blue
    elif 200 < self.effect_length < 250:
        secondary_color = dull_orange
    elif 150 < self.effect_length < 200:
        secondary_color = dull_green
    elif 100 < self.effect_length < 150:
        secondary_color = dull_yellow
    elif 50 < self.effect_length < 100:
        secondary_color = dull_pink
    elif 0 < self.effect_length < 50:
        secondary_color = dull_red
```



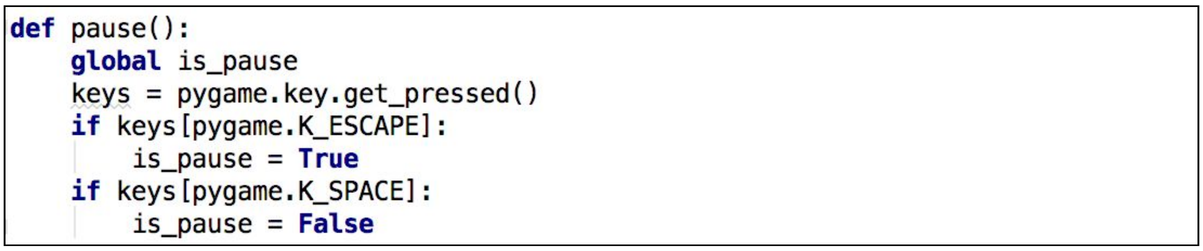
```
# after rainbow land, the bg color should return to its previous set color
if spin == 2:
    secondary_color = previous_s_color
arcade.reset_effects()
```



```
def score_collision(self):
    if ball.ball_pos[0] <= ball.ball_radius:
        self.p2_score += 1
        score.arcade_score()
        ball.start_game()
    elif ball.ball_pos[0] >= game.screen_width + ball.ball_radius:
        self.p1_score += 1
        score.arcade_score()
        ball.start_game()

@staticmethod
def arcade_score():
    global secondary_color
    if mode == 2:
        # if 'rainbow land' power up is active and player scores, the bg color should reset to previous
        if spin == 2:
            secondary_color = previous_s_color
        arcade.reset_arcade()
```

For rainbow land, I used timing to change the background colors. I also saved the ‘previous’ background so that it can be returned to it after the power-up ends or a player scores.

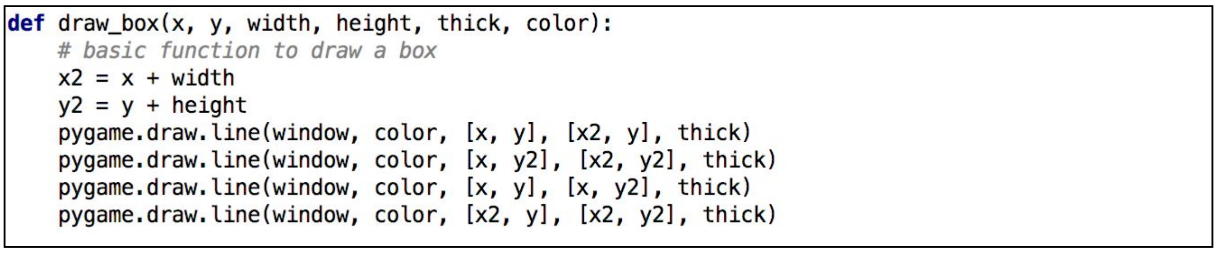
Pause Function



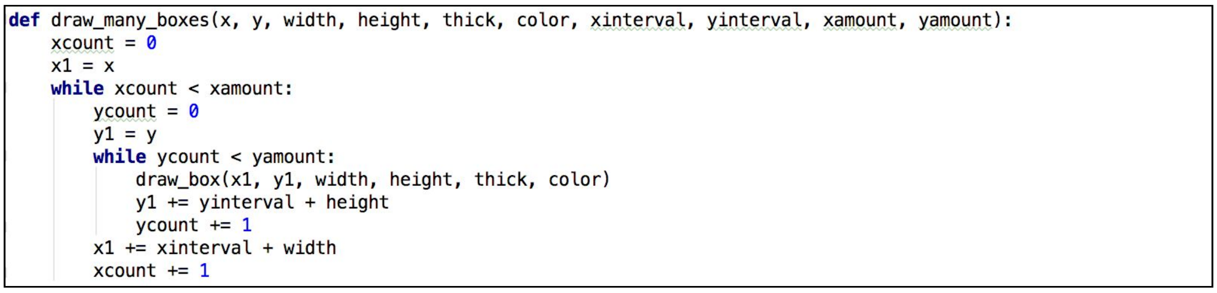
```
def pause():
    global is_pause
    keys = pygame.key.get_pressed()
    if keys[pygame.K_ESCAPE]:
        is_pause = True
    if keys[pygame.K_SPACE]:
        is_pause = False
```

For the pause function, I again used boolean logic that will make sense in my game loop.

The GUI boxes



```
def draw_box(x, y, width, height, thick, color):
    # basic function to draw a box
    x2 = x + width
    y2 = y + height
    pygame.draw.line(window, color, [x, y], [x2, y], thick)
    pygame.draw.line(window, color, [x, y2], [x2, y2], thick)
    pygame.draw.line(window, color, [x, y], [x, y2], thick)
    pygame.draw.line(window, color, [x2, y], [x2, y2], thick)
```

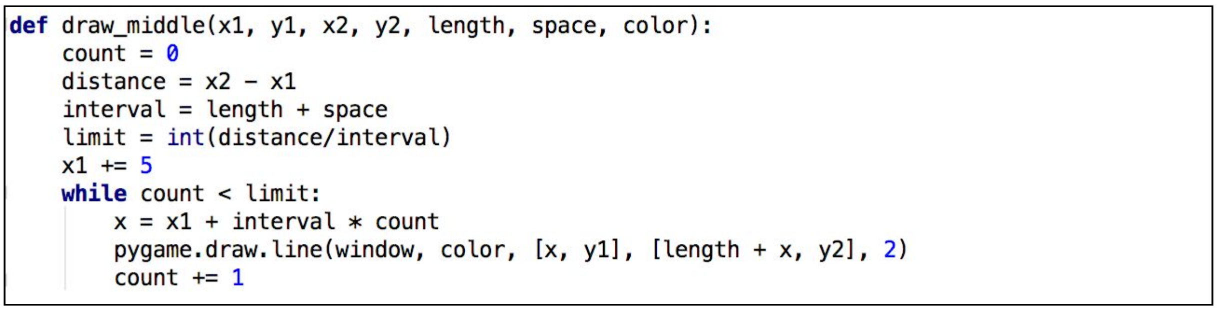


```
def draw_many_boxes(x, y, width, height, thick, color, xinterval, yinterval, xamount, yamount):
    xcount = 0
    x1 = x
    while xcount < xamount:
        ycount = 0
        y1 = y
        while ycount < yamount:
            draw_box(x1, y1, width, height, thick, color)
            y1 += yinterval + height
            ycount += 1
        x1 += xinterval + width
        xcount += 1
```

To draw a box, I used four lines and mathematically linked them.

To draw several boxes under a constant interval, which I used in my ‘settings’ section, I used a nested loop to continuously draw boxes at a set interval.

The GUI Middleline

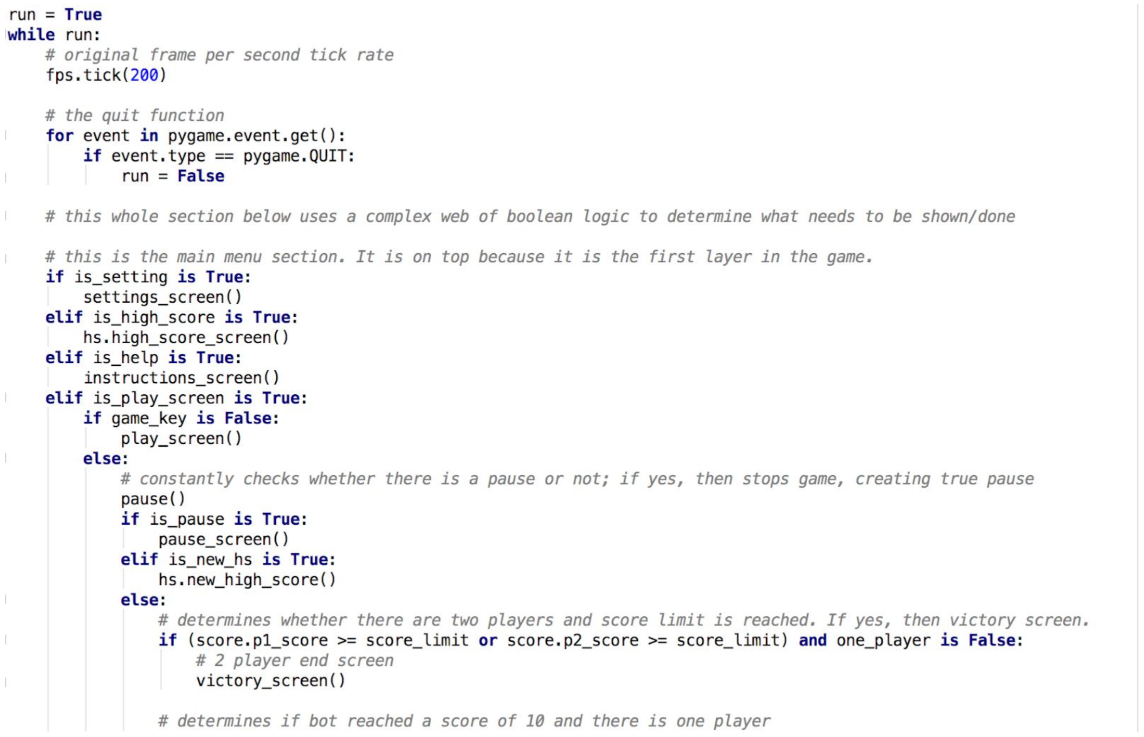


```python
def draw_middle(x1, y1, x2, y2, length, space, color):
    count = 0
    distance = x2 - x1
    interval = length + space
    limit = int(distance/interval)
    x1 += 5
    while count < limit:
        x = x1 + interval * count
        pygame.draw.line(window, color, [x, y1], [length + x, y2], 2)
        count += 1
```

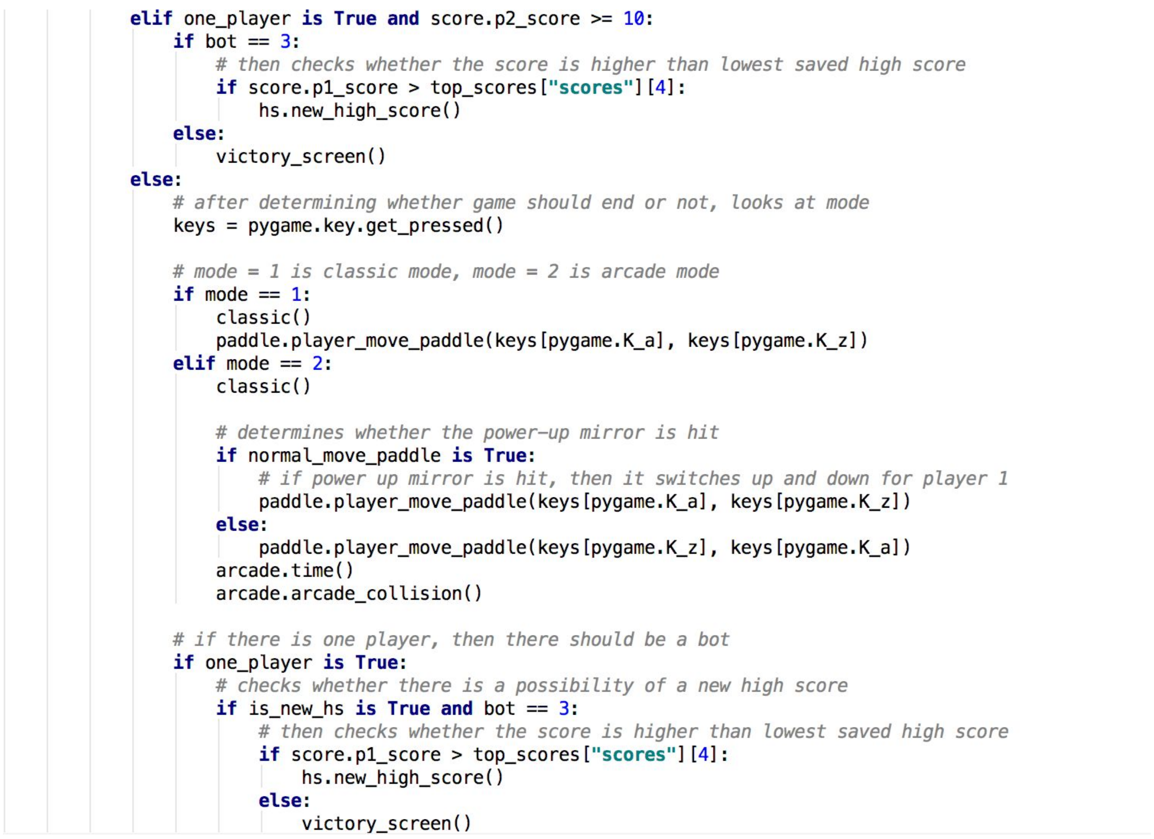
For the dashed middle line, I first determine how many dashes I need by determining the distance and then dividing by the length of the lines and the space in between. Then, I draw dashes until that limit is reached.

The Continuous Game Loop (Heart of the Code)

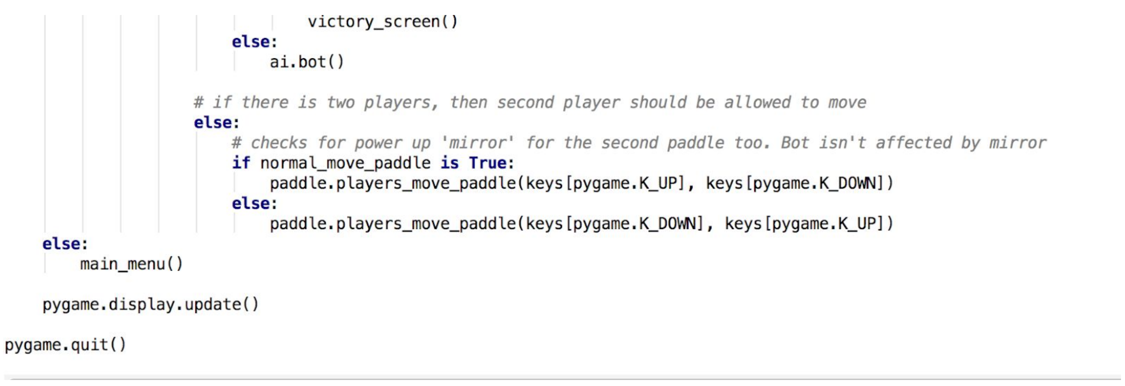
For the game loop connecting all my functions, I used complex boolean logic to determine where the user needs to be at. It all results in this:



```python
run = True
while run:
    # original frame per second tick rate
    fps.tick(200)

    # the quit function
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            run = False

    # this whole section below uses a complex web of boolean logic to determine what needs to be shown/done

    # this is the main menu section. It is on top because it is the first layer in the game.
    if is_setting is True:
        settings_screen()
    elif is_high_score is True:
        hs.high_score_screen()
    elif is_help is True:
        instructions_screen()
    elif is_play_screen is True:
        if game_key is False:
            play_screen()
        else:
            # constantly checks whether there is a pause or not; if yes, then stops game, creating true pause
            pause()
            if is_pause is True:
                pause_screen()
            elif is_new_hs is True:
                hs.new_high_score()
            else:
                # determines whether there are two players and score limit is reached. If yes, then victory screen.
                if (score.p1_score >= score_limit or score.p2_score >= score_limit) and one_player is False:
                    # 2 player end screen
                    victory_screen()

                # determines if bot reached a score of 10 and there is one player
```



```
            elif one_player is True and score.p2_score >= 10:
                if bot == 3:
                    # then checks whether the score is higher than lowest saved high score
                    if score.p1_score > top_scores["scores"][4]:
                        hs.new_high_score()
                else:
                    victory_screen()
            else:
                # after determining whether game should end or not, looks at mode
                keys = pygame.key.get_pressed()

                # mode = 1 is classic mode, mode = 2 is arcade mode
                if mode == 1:
                    classic()
                    paddle.player_move_paddle(keys[pygame.K_a], keys[pygame.K_z])
                elif mode == 2:
                    classic()

                    # determines whether the power-up mirror is hit
                    if normal_move_paddle is True:
                        # if power up mirror is hit, then it switches up and down for player 1
                        paddle.player_move_paddle(keys[pygame.K_a], keys[pygame.K_z])
                    else:
                        paddle.player_move_paddle(keys[pygame.K_z], keys[pygame.K_a])
                    arcade.time()
                    arcade.arcade_collision()

                # if there is one player, then there should be a bot
                if one_player is True:
                    # checks whether there is a possibility of a new high score
                    if is_new_hs is True and bot == 3:
                        # then checks whether the score is higher than lowest saved high score
                        if score.p1_score > top_scores["scores"][4]:
                            hs.new_high_score()
                        else:
                            victory_screen()
```



```
                            victory_screen()
                    else:
                        ai.bot()

                # if there is two players, then second player should be allowed to move
                else:
                    # checks for power up 'mirror' for the second paddle too. Bot isn't affected by mirror
                    if normal_move_paddle is True:
                        paddle.players_move_paddle(keys[pygame.K_UP], keys[pygame.K_DOWN])
                    else:
                        paddle.players_move_paddle(keys[pygame.K_DOWN], keys[pygame.K_UP])
    else:
        main_menu()

    pygame.display.update()

pygame.quit()
```

To sum it all up, it can be divided into three layers: main menu, play screen, and the actual game.

1022 Words (excluding headers)