```
### fuctions
# this is the function used when the ball is clicked, it returns to the starting spot based on which map
def return_home(x, y):
    global which_map
    if which_map == "m_one":
        move(ball, -300, -200)
    if which_map == "m_two":
        move(ball, 250, 200)
    if which_map == "m_three":
        move(ball, 0, -200)
    if which_map == "m_four":
        move(ball, 0, -100)
```

```
map_list = ["m_one", "m_two", "m_three", "m_four"]
which_map = rand.choice(map_list)
```

Together, these two sections work together to make sure that the ball always returns back to it's starting point if the user ever wants it to do so. The map that the game chooses for you is a ¼ decision out of the four maps in the map list. The "return home" function takes the value of "which_map", and based on what the value is returns the ball to a spot on the grid. This spot on the grid is the starting location for each level, as it is different for each of them. You can see that the "return home" function actually has two parameters, and these variables take in the location of a click, as this is a function that is used by an "onclick" method.

```
def sand_check():
    global which_map
    if which_map == "m_one" or "m_two" or "m_three":
        sand_list = [sand_one, sand_two]
    else:
        sand_list = [sand_one, sand_two, sand_three, sand_four]

    for turtles in sand_list:
        if ball.xcor() <= turtles.xcor() + 50 and ball.xcor() >= turtles.xcor() - 50:
            if ball.ycor() <= turtles.ycor() + 50 and ball.ycor() >= turtles.ycor() - 50:
                update_score()
```

```
def pond_check():
    global which_map
    if which_map == "m_two" or "m_three" or "m_four":
        pond_list = [pond_one, pond_two]
    else:
        pond_list = [pond_one]

    for turtles in pond_list:
        if ball.xcor() <= turtles.xcor() + 50 and ball.xcor() >= turtles.xcor() - 50:
            if ball.ycor() <= turtles.ycor() + 50 and ball.ycor() >= turtles.ycor() - 50:

                if which_map == "m_one":   # this is just the "return home" function, but wi
                    move(ball, -300, -200)
                if which_map == "m_two":
                    move(ball, 250, 200)
                if which_map == "m_three":
                    move(ball, 0, -200)
                if which_map == "m_four":
                    move(ball, 0, -100)
```

The value of “which_map” is once again vital in these two functions. “sand_check” and “pond_check” are the two functions that find out if the ball is in sand or a pond respectively. To do so, the game first needs to find out how many sand dunes or ponds it needs to check through - so it refers back to the “which_map” variable. The levels all have different amounts of obstacles, and we know how many obstacles each of the levels has. Depending on the value of “which_up” the game creates two lists of all the sand and pond turtles it needs to check through. Then these two lists are looped through, with the loop variable being the turtle in question. Both the current x and y coordinates of the ball and “turtle” are checked, and since the obstacles aren’t a pixel wide and long, a 50 pixel radius around the obstacle is checked as well. If both of the conditionals come out to be true, then a different action is executed. If the ball hits sand, then an additional hit is added to the score using the “update score” function (documented later), and if the ball hits falls in a pond, it goes back to the levels start. You may recognize this as a copy of the “return_home” function, which would be correct. However that function takes two parameters, which aren’t needed here, resulting in the content of the function being repeated here.

```
def update_score():
    global score
    score += 1
    score_writer.clear()
    score_writer.write("Score: " + str(score), font=font_setup)
```

```
def golf_hit():
    ball.penup()
    power_check = 0

    while power_check == 0: # this section will repeat until the numbers given by the user correctly allig
        power_and_direction = input("How much power (1-10)?, What direction (0-359) (0 = East, 90 = North,
        power_and_direction = power_and_direction.split(":")
        if int(power_and_direction[0]) < 11 and int(power_and_direction[0]) > 0:
            if int(power_and_direction[1]) < 360 and int(power_and_direction[1]) > -1:
                power_check = 1

    ball.seth(int(power_and_direction[1]))
    ball.fd(int(power_and_direction[0])*40)
    sand_check() # the sand check and pond check (later created) are both intregrated into the golf hit fu
    pond_check()
    update_score()
```

Right away, you can see that this is where both the “sand_check” and “pond_check” functions are integrated. The “update_score” function is very simple. “Score” and “score_writer” were priorly defined, with “score” receiving a value of 0, and “score_writer” being initialized as a turtle and moving to the top left of the window. Every time this function is played through, 1 is added to the score, the “score_writer” erases everything that it has written, and writes up the new score. This function ends up being integrated in the “golf_hit” function, right after the two

obstacle checks. For the actual “golf_hit” function, it first asks the user how much power and what direction they want to send it in at once. The power is put to a multiplier of 40, and the direction is directly placed as the “setheading” of the ball. This is properly explained within the question, hence it going off the screen. The question asks the user to format it like [Power:Direction], which would look something like 6:150 in practice. The two conditionals are in the case of the user entering something silly such as 99999:99999, 20:2, 1:400, etc. You can see that a while loop is placed which goes on until “power_check” is more than 0. After the code confirms that the numbers are indeed viable, it sets “power_check” to 1, hence ending the loop. If the numbers weren’t correct, it would ask the question over and over until it would be. After that, the function is self explanatory, setting itself to the user’s angle and moving the amount they want them two.

```
def draw_map_four():  # start of map four

    draw.color("#E40010")

    move(ball, 0, -100)
    move(hole, 0, 100)
    move(sand_one, 200, -40)
    move(sand_two, 70, 180)
    move(sand_three, -230, -100)
    move(sand_four, -100, 25)
    move(pond_one, -240, 180)
    move(pond_two, 240, -160)
    move(draw, 0, 0)


    draw.rt(45)
    for i in range(2):
        draw.fd(125)
        draw.bk(250)
        draw.fd(125)
        draw.rt(90)
```

```
    move(draw, -350, -220)
    draw.rt(45)
    for i in range(2):
        draw.fd(225)
        draw.rt(90)
        draw.fd(75)
        draw.bk(75)
        draw.lt(90)
        draw.fd(225)

        draw.rt(90)
        draw.fd(700)
        draw.rt(90)

    draw.hideturtle()
```

The maps are each a long process to create, so only one of them is being documented here. This is the fourth map, which is a large rectangle with an “X” in between. The second image shown here is just a continuation of the first. They are both one function. In the beginning the “move” function is used, which is super simple, only consisting of “penup,” “goto,” and “pendown.” The three parameters are the turtle in question, and the x/y coordinates. The move function is being used to put the obstacles into place. Next, the drawer turns 45 degrees and goes through a loop in which it moves forwards, backwards, forwards again, and then turns. This is creating the “X” in

the center. The “move” function is once again used to get the drawer in position for the large rectangle, which is a very simple for loop as all rectangles are. However there is a barrier in the center of the short sides that reaches inward toward the “X.” Halfway through drawing the short side, the turtle turns right, moves forward and backward, turns left, and continues as usual. At the very end of the function, the turtle hides itself to not distract the player.

```
countdown() # starting the timer, until the timer ends you can hit the ball and reset it

while timer_up == False:
    golf_hit()
    ball.onclick(return_home)

    if ball.xcor() <= hole.xcor() + 30 and ball.xcor() >= hole.xcor() - 30:
        if ball.ycor() <= hole.ycor() + 30 and ball.ycor() >= hole.ycor() - 30:
            counter.penup(), counter.clear, counter.hideturtle()
            score_writer.clear()
            score_writer.write("You Win! Score: " + str(score), font=font_setup)
            timer = 0
```

This is the main code segment that runs the entire game. You will see a “countdown” function that is the exact same function used in “1.2.1 Catch a Turtle” to set a 30 second timer. The only changes made are that the timer is now 100 seconds long. The while loop is based on “timer_up” being false, which it is until the 100 seconds are up. The “golf_hit” and “return_home” functions should be familiar, as they were earlier discussed. The two conditionals are also very familiar, as it is the exact same code used in “sand_check” and “pond_check” to see if the ball’s x/y coordinates match with the obstacles’. Although this time there is a radius of 60 rather than 100, as getting the ball in the hole should be more challenging. Once the ball is in the hole, the “counter” will completely disappear, and the “score_writer” will additionally display “You Win!” on top of the score. The timer will be set to 0, and the game will be over.