Lesson 11: Turtle Minefield Game

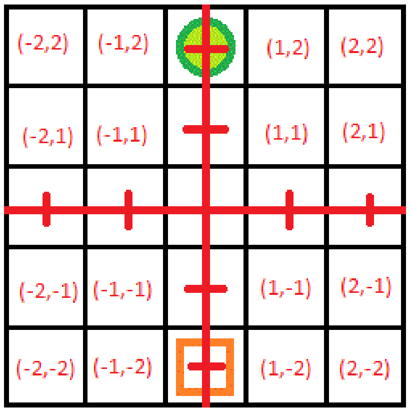
In this lesson, we'll be making another game with our turtle skills and be using a grid as a way to control our turtle positions.

The point of the game is to move from the start to the finish without running into hidden mines placed randomly.

PART 1: Setting up the Turtle and Grid Scene

Open the file called "turtle_minefield_student.py" and you will see many lines of code already written for you broken into six parts.

Review the code. Part one will have code that looks familiar from earlier lessons. It will start by importing the turtle module. The player turtle (Tom) is created and the pen color is made red. We set the background of the screen to our image called "obstacle_course.gif" that shows a grid with a green circle (the finish goal) at the top center and an orange square (the starting position) at the bottom center. Our grid is made of 5x5 squares that are 50 pixels in size. We will be referring to each square based off the (x, y) positions that have an origin at (0, 0) in the center square and will be increasing/decreasing by 1 for each square. Look below to get an idea of how the positions are referenced.



Part one of the code then creates an empty array called obstacles where we will be storing each of five hidden mine positions in the next part.
We will also ,create a variable to hold the number of tries (named tries) and another variable to hold our grid size (named gridsize) with a value of 50 pixels.

```
obstacles = []
tries = 3
gridsize = 50
```

We then set two variables - our starting position (pos) and the finish (goal) to a tuple.

```
pos = (0, -2)
goal = (0, 2)
```

The final section of part one will print the game information and instructions to play.

PART 2: Placing the Hidden Mines Randomly

In this part, we write code to place five mines in random grid positions, as long as it's not on the starting or finishing position or a position where a mine already exists.

We will start with a for loop which will repeat 5 times - one for each mine that needs to be placed.

Then, we will create a new tuple called mine and assign it to the player position. This assignment is temporary and will change shortly.

Next, we create a while loop which will continue until mine (the mine position) is in a valid position. Again, the mine is not valid if it is on the starting position (which we initially assigned it to) or if it's on the goal position or if the position is already added to list of previous mine positions.

Try adding this code below the PART TWO comment:

```
# Place five obstacles in the grid at random positions

for i in range(5):

    mine = pos

    while(mine == pos or mine == goal or mine in obstacles):

        minex = random.randint(-2, 2)

        miney = random.randint(-2, 2)

        mine = (minex, miney)

    obstacles.append(mine)
```

Note that, inside the while loop, we assign two more temporary variables for the x and y coordinates of the potential mine location. Then we reassign the temp variable to a new tuple with the random grid position.

If we make it outside the while loop, then that means we have a valid mine position. We can now add it to the obstacles list and continue with the for loop.

(mine in obstacles) will return True if the item, mine, is already inside the list, obstacles. This function is useful for making sure there are no duplicates in an array.

PART 3: Moving our Turtle

Here, we'll make a simple function that we can use to move any turtle object to a specific position on our grid.

Put in the missing lines of code under the PART THREE comments:

```
. . .
# Move the turtle to a specific grid cell
def moveto(coordinates, T):
    T.goto(coordinates[0] * gridsize, coordinates[1] * gridsize)
```

The function will take in a tuple called coordinates and a turtle object called T.

The coordinates will be a position in our grid, so we need to take into account each grid square is 50 pixels in size. We need to multiply the grid positions by the grid size in order for the grid position to be equal to the actual screen position.

We then call the turtle (T) function goto() so that our turtle will move to the (x,y) position given.

This function will save us from writing this long piece of code every time we need to move the player or mine hit.

PART 4: Creating an X when Player Hits a Mine

In this part, we will be making a function to draw a red X at a given location.

This will be used later when the player hits a hidden mine.

The function will take a tuple for the coordinates of where we want the X to be drawn.

We start this function by creating a new turtle called hit which will draw the red X. We only want to see what the turtle draws so we hide the turtle.

The given code is missing one line that you will need to figure out.

We need a line of code that uses our moveto() function that we created in the previous part. It will need to move our hit turtle to the given coordinates.

Fill in the blank with missing line of code:

```
. . .
hit.pencolor("red")
hit.speed(0)
hit.penup()
____ ____________________
hit.setheading(45)
hit.fd(25)
hit.rt(180)
```

If you get stumped, try looking at the rest of the code to see another example of us using the moveto() function.

After we move the hit object to the specific grid square, we then begin to draw an X by using the forward and turning commands.

Now we are done with this function. Below the function, you will see code that moves our turtle Tom to the starting position, makes him face upward, and puts the pen down ready to draw. Remember, **setheading(degrees)** is a turtle method that will turn the turtle counter-clockwise the given amount of degrees, where 0 degrees is facing right.

PART 5: Taking User Input to Move the Player

This section will cover taking input as commands for moving the player. This is very similar to our obstacle game except we move in all directions and rotate to face that direction.

We begin our game state while loop that will end once we reach the goal, quit the game, or lose the game by running out of tries.

We take in the input and set two temporary variables for the x and y coordinate of Tom.

Then we begin our if statements to see what the user input and if it is a valid move.

For each if statement for the direction commands, we set the heading to face the right direction and then change the temporary coordinate variable.

The up and left if statements are missing lines of code to make them function correctly.

Use the complete right and down if statements as reference to what they are missing.

Fill in the two missing lines of code:

```
. . .
if(command == "right" and x < 2):
    Tom.setheading(0)
    x += 1
elif(command == "up" and y < 2):
    ____________________
    y += 1
elif(command == "left" and x > -2):
    Tom.setheading(180)
    ____________________
elif(command == "down" and y > -2):
    Tom.setheading(270)
    y -= 1
```

After these if statements are two more for quitting the game and printing a message if it's not a valid command.

Then, we set Tom's position to the new x and y coordinates, and proceed to move him there by using our moveto() function again.

Test your code at this point by running the module. You should be able to move left, up, down, right. Try typing each of the commands to make sure they work.

If they all work then try moving to the finish goal (the green circle). It should close the screen and print out a victory message.

PART 6: Checking for a Collision Between the Turtle and a Mine

In this part we need to check if the player is on a mine, and, if so, draw a red X, subtract a try. Then check if they are out of tries (end game) or restart player at start.

Add the following code below the PART SIX comment:

```
    ###PART SIX###
    """
    if (pos in obstacles):
        print("You hit an obstacle!")
        showobstacle(pos)
        tries -= 1
        if(tries == 0):
            break
        else:
            Tom.penup()
            pos = (0, -2)
            moveto(pos, Tom)
            Tom.setheading(90)
            Tom.pendown()
```

So we start with using the “in” function to check if the player position is equal to one of the mine positions held in the list obstacles.

If that is true, then we have hit an obstacle. We call our function we made earlier to show the obstacle at the player position, which will draw the red X.

Then we subtract a try from our total. If we are out of tries, then break out of the game state while loop because we have lost.

If we have another try, then we need to reset our player. Pick the pen up, make position equal to start, move player to start, rotate to face upwards, and then place the pen back down. Test the code out and see your player resets after hitting a mine. Then try hitting a mine three times. The screen should close and a “Game Over” message should appear. If everything works, the game is now complete!