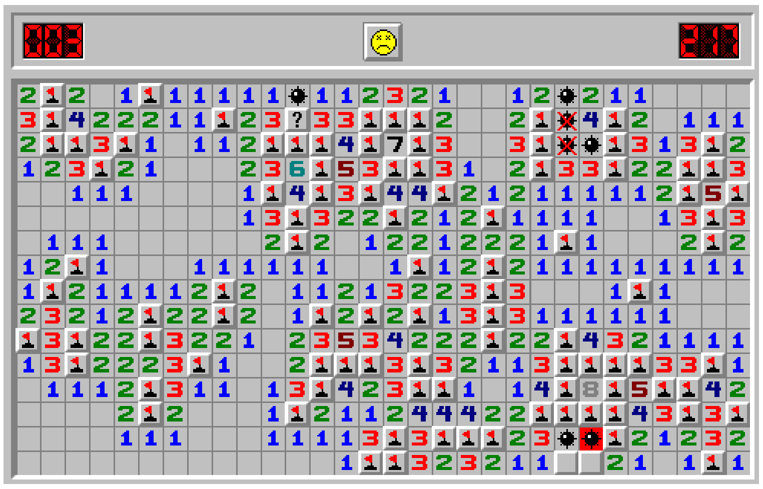
Project report On Minesweeper

Submitted by:

Name	**: Siripireddy Bharath Bhushan Reddy**
Registration Number	**:11909164**
Section	**:K19GT**
Roll Number	**:49**

Introduction



My project is on Minesweeper, It is a single-player game in which the player has to clear a square grid containing mines and numbers. The player has to prevent himself from landing on a mine with the help of numbers in the neighbouring tiles.

Modules

pyminesweeper

- MinesweeperMap
  - Contains the functions to create the minesweeper grid and help connect to a frontend
- MinesweeperUI
  - Contains terminal UI for playing the game and functions to create a customised game UI

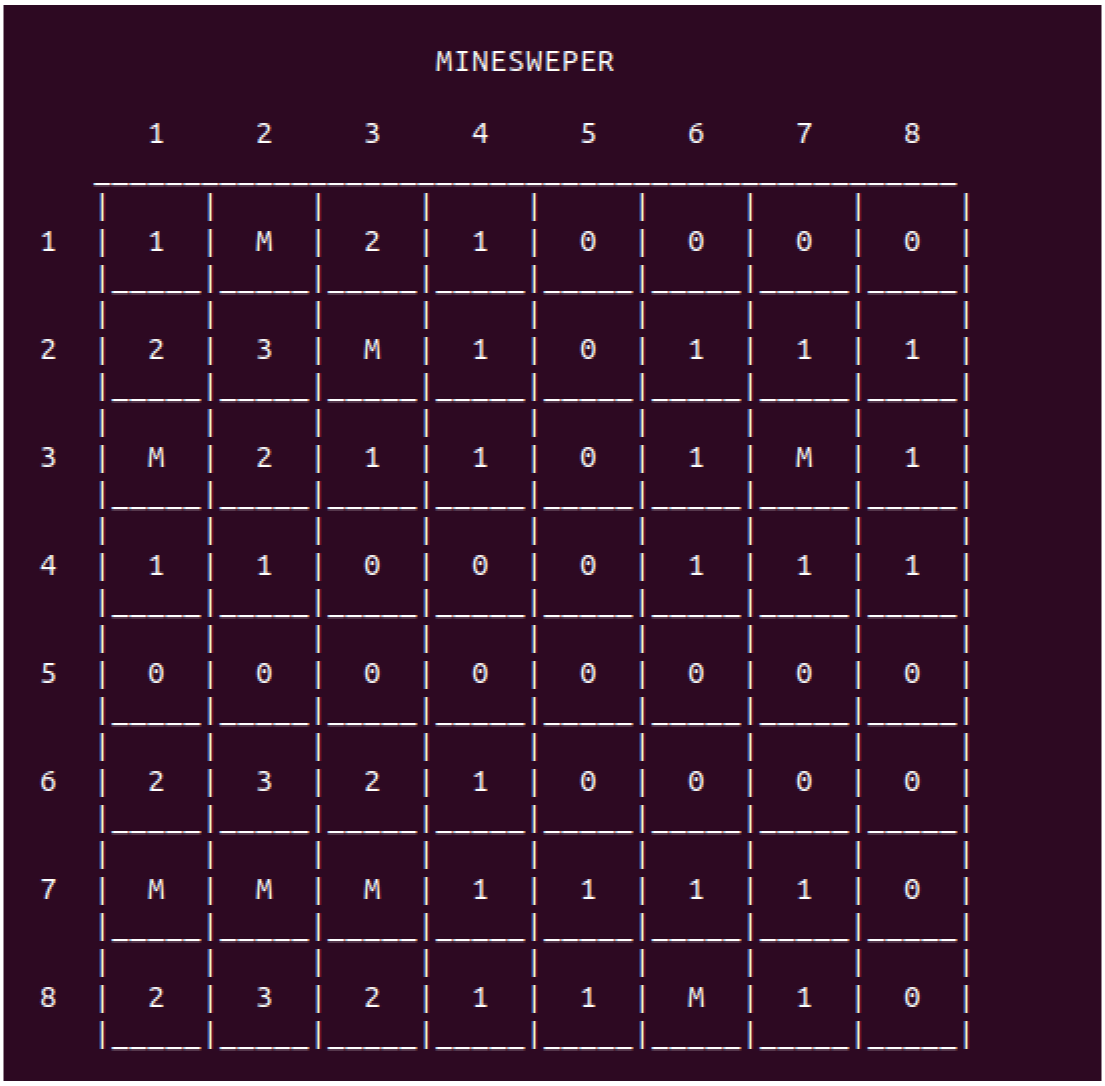
Designing Minesweeper Using Python

Before creating the game logic, we need to design the basic layout of the game. A square grid is rather easy to create using Python by:

```
# Printing the Minesweeper Layout
def print_mines_layout():
    global mine_values
    global n

    print()
    print("\t\t\tMINESWEEPER\n")
```

```
st = "   "
for i in range(n):
    st = st + "      " + str(i + 1)
print(st)

for r in range(n):
    st = "      "
    if r == 0:
        for col in range(n):
            st = st + "_______"
        print(st)

    st = "      "
    for col in range(n):
        st = st + "|      "
    print(st + "|")

    st = "  " + str(r + 1) + "   "
    for col in range(n):
        st = st + "|   " + str(mine_values[r][col]) + "   "
    print(st + "|")

    st = "      "
    for col in range(n):
        st = st + "|______"
    print(st + '|')

print()
```

The grid displayed in each iteration resembles the following figure:



```
                        MINESWEEPER

        1     2     3     4     5     6     7     8
     ________________________________________________
     |     |     |     |     |     |     |     |     |
  1  |  1  |  M  |  2  |  1  |  0  |  0  |  0  |  0  |
     |_____|_____|_____|_____|_____|_____|_____|_____|
     |     |     |     |     |     |     |     |     |
  2  |  2  |  3  |  M  |  1  |  0  |  1  |  1  |  1  |
     |_____|_____|_____|_____|_____|_____|_____|_____|
     |     |     |     |     |     |     |     |     |
  3  |  M  |  2  |  1  |  1  |  0  |  1  |  M  |  1  |
     |_____|_____|_____|_____|_____|_____|_____|_____|
     |     |     |     |     |     |     |     |     |
  4  |  1  |  1  |  0  |  0  |  0  |  1  |  1  |  1  |
     |_____|_____|_____|_____|_____|_____|_____|_____|
     |     |     |     |     |     |     |     |     |
  5  |  0  |  0  |  0  |  0  |  0  |  0  |  0  |  0  |
     |_____|_____|_____|_____|_____|_____|_____|_____|
     |     |     |     |     |     |     |     |     |
  6  |  2  |  3  |  2  |  1  |  0  |  0  |  0  |  0  |
     |_____|_____|_____|_____|_____|_____|_____|_____|
     |     |     |     |     |     |     |     |     |
  7  |  M  |  M  |  M  |  1  |  1  |  1  |  1  |  0  |
     |_____|_____|_____|_____|_____|_____|_____|_____|
     |     |     |     |     |     |     |     |     |
  8  |  2  |  3  |  2  |  1  |  1  |  M  |  1  |  0  |
     |_____|_____|_____|_____|_____|_____|_____|_____|
```

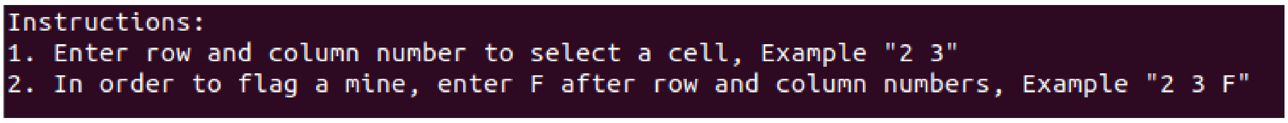
The `'M'` symbol denotes the presence of a 'mine' in that cell. As we can see clearly, any number on the grid denotes the number of mines present in the neighbouring 'eight' cells.

The use of variables like, `mine_values` will be explained further in the tutorial.

Input system

One of the most important parts of any game is sustaining the input method. In our version of Minesweeper, we will be using the row and column numbers for our input technique.

Before starting the game, the script must provide a set of instructions for the player. Our game prints the following.



```
Instructions:
1. Enter row and column number to select a cell, Example "2 3"
2. In order to flag a mine, enter F after row and column numbers, Example "2 3 F"
```

Minesweeper Instructions

The row and column numbers displayed along with the grid are helpful for our input system. As we know, keeping track of mines without any indicator can be difficult. Therefore, Minesweeper has a provision of using ‘flag’ to mark the cells, which we know contains a mine.

Data Storage

For a single game of Minesweeper, we need to keep track of the following information:

- The **size** of the grid.
- The **number of mines**.
- **The ‘actual’ grid values** – At the start of the game, we need a container for storing the real values for the game, unknown to the player. For instance, the location of mines.
- **The ‘apparent’ grid values** – After each move, we need to update all the values that must be shown to the player.
- **The flagged positions** – The cells which have been flagged.

These values are stored using the following data structures

```
if __name__ == "__main__":

    # Size of grid
    n = 8
    # Number of mines
    mines_no = 8

    # The actual values of the grid
    numbers = [[0 for y in range(n)] for x in range(n)]
    # The apparent values of the grid
    mine_values = [[' ' for y in range(n)] for x in range(n)]
    # The positions that have been flagged
    flags = []
```

There is not much in the game-logic of Minesweeper. All the effort is to be done in setting up the Minesweeper layout.

Setting up the Mines

We need to set up the positions of the mines randomly, so that the player might not predict their positions. This can be done by:

```
# Function for setting up Mines
def set_mines():

    global numbers
    global mines_no
    global n

    # Track of number of mines already set up
    count = 0
    while count < mines_no:

        # Random number from all possible grid positions
        val = random.randint(0, n*n-1)

        # Generating row and column from the number
        r = val // n
        col = val % n

        # Place the mine, if it doesn't already have one
        if numbers[r][col] != -1:
            count = count + 1
            numbers[r][col] = -1
```

In the code, we choose a random number from all possible cells in the grid. We keep doing this until we get the said number of mines.

Note: The actual value for a mine is stored as -1, whereas the values stored for display, denote the mine as 'M'.

Note: The ‘randint’ function can only be used after importing the random library. It is done by writing 'import random' at the start of the program.

Setting up the grid numbers

For each cell in the grid, we have to check all adjacent neighbours whether there is a mine present or not. This is done by:

```
# Function for setting up the other grid values
def set_values():


    global numbers
    global n


    # Loop for counting each cell value
    for r in range(n):
        for col in range(n):


            # Skip, if it contains a mine
            if numbers[r][col] == -1:
                continue


            # Check up
            if r > 0 and numbers[r-1][col] == -1:
                numbers[r][col] = numbers[r][col] + 1
            # Check down
            if r < n-1 and numbers[r+1][col] == -1:
                numbers[r][col] = numbers[r][col] + 1
            # Check left
            if col > 0 and numbers[r][col-1] == -1:
                numbers[r] = numbers[r] + 1
            # Check right
            if col < n-1 and numbers[r][col+1] == -1:
                numbers[r][col] = numbers[r][col] + 1
            # Check top-left
            if r > 0 and col > 0 and numbers[r-1][col-1] == -1:
                numbers[r][col] = numbers[r][col] + 1
            # Check top-right
            if r > 0 and col < n-1 and numbers[r-1][col+1]== -1:
                numbers[r][col] = numbers[r][col] + 1
            # Check below-left
            if r < n-1 and col > 0 and numbers[r+1][col-1]== -1:
                numbers[r][col] = numbers[r][col] + 1
            # Check below-right
            if r < n-1 and col< n-1 and numbers[r+1][col+1]==-1:
                numbers[r][col] = numbers[r][col] + 1
```

These values are to be hidden from the player, therefore they are stored in numbers variable.

Game Loop

Game Loop is a very crucial part of the game. It is needed to update every move of the player as well as the conclusion of the game.

```
# Set the mines
set_mines()


# Set the values
set_values()


# Display the instructions
```

```
instructions()


# Variable for maintaining Game Loop
over = False


# The GAME LOOP
while not over:
    print_mines_layout()
```

In each iteration of the loop, the Minesweeper grid must be displayed as well as the player's move must be handled.

Handle the player input

As we mentioned before, there are two kinds of player input :

```
# Input from the user
inp = input("Enter row number followed by space and column number = ").split()
```

Standard input

In a normal kind of move, the row and column number are mentioned. The player's motive behind this move is to unlock a cell that does not contain a mine.

```
# Standard Move
if len(inp) == 2:


    # Try block to handle errant input
    try:
        val = list(map(int, inp))
    except ValueError:
        clear()
        print("Wrong input!")
        instructions()
        continue
```

Flag input

In a flagging move, three values are sent in by the gamer. The first two values denote cell location, while the last one denotes flagging.

```
# Flag Input
elif len(inp) == 3:
    if inp[2] != 'F' and inp[2] != 'f':
        clear()
        print("Wrong Input!")
        instructions()
        continue

    # Try block to handle errant input
    try:
        val = list(map(int, inp[:2]))
    except ValueError:
        clear()
        print("Wrong input!")
        instructions()
        continue
```

Sanitize the input

After storing the input, we have to do some sanity checks, for the smooth functioning of the game.

```
# Sanity checks
if val[0] > n or val[0] < 1 or val[1] > n or val[1] < 1:
    clear()
    print("Wrong Input!")
    instructions()
    continue

# Get row and column numbers
r = val[0]-1
```

```
col = val[1]-1
```

On the completion of input process, the row and column numbers are to be extracted and stored in 'r' and 'c'.

Handle the flag input

Managing the flag input is not a big issue. It requires checking for some pre-requisites before flagging the cell for a mine.

The following checks must be made:

- The cell has already been flagged or not.
- Whether the cell to be flagged is already displayed to the player.
- The number of flags does not exceed the number of mines.

After taking care of these issues, the cell is flagged for a mine.

```
# If cell already been flagged
if [r, col] in flags:
    clear()
    print("Flag already set")
    continue

# If cell already been displayed
if mine_values[r][col] != ' ':
    clear()
    print("Value already known")
    continue

# Check the number for flags
if len(flags) < mines_no:
    clear()
    print("Flag set")

    # Adding flag to the list
    flags.append([r, col])


    # Set the flag for display
    mine_values[r][col] = 'F'
    continue
else:
    clear()
    print("Flags finished")
    continue
```

Handle the standard input

The standard input involves the overall functioning of the game. There are three different scenarios:

Anchoring on a mine

The game is finished as soon as the player selects a cell having a mine. It can happen out of bad luck or poor judgment.

```
# If landing on a mine --- GAME OVER
if numbers[r][col] == -1:
    mine_values[r][col] = 'M'
    show_mines()
    print_mines_layout()
    print("Landed on a mine. GAME OVER!!!!!")
    over = True
    continue
```

After we land on a cell with mine, we need to display all the mines in the game and alter the variable behind the game loop.

The function 'show_mines()' is responsible for it.

```
def show_mines():
    global mine_values
    global numbers
    global n
    for r in range(n):
        for col in range(n):
            if numbers[r][col] == -1:
                mine_values[r][col] = 'M'
```

Visiting a ‘0’-valued cell.

The trickiest part of creating the game is managing this scenario. Whenever a gamer, visits a ‘0’-valued cell, all the neighboring elements must be displayed until a non-zero-valued cell is reached.

```
# If landing on a cell with 0 mines in neighboring cells
elif numbers[r][n] == 0:
    vis = []
    mine_values[r][n] = '0'
    neighbours(r, col)
```

This objective is achieved using **Recursion**. Recursion is a programming tool in which the function calls itself until the base case is satisfied.
The neighbours function is a recursive one, solving our problem.

```
def neighbours(r, col):

    global mine_values
    global numbers
    global vis

    # If the cell already not visited
    if [r,col] not in vis:

        # Mark the cell visited
        vis.append([r,col])

        # If the cell is zero-valued
        if numbers[r][col] == 0:

            # Display it to the user
            mine_values[r][col] = numbers[r][col]

            # Recursive calls for the neighbouring cells
            if r > 0:
                neighbours(r-1, col)
            if r < n-1:
                neighbours(r+1, col)
            if col > 0:
                neighbours(r, col-1)
            if col < n-1:
                neighbours(r, col+1)
            if r > 0 and col > 0:
                neighbours(r-1, col-1)
            if r > 0 and col < n-1:
                neighbours(r-1, col+1)
            if r < n-1 and col > 0:
                neighbours(r+1, col-1)
            if r < n-1 and col < n-1:
                neighbours(r+1, col+1)

        # If the cell is not zero-valued
        if numbers[r][col] != 0:
                mine_values[r][col] = numbers[r][col]
```

For this particular concept of the game, a new data structure is used, namely, vis. The role of vis to keep track of already visited cells during recursion. Without this information, the recursion will continue perpetually.

After all the cells with zero value and their neighbours are displayed, we can move on to the last scenario.

Choosing a non zero-valued cell

No effort is needed to handle this case, as all we need to do is alter the displaying value.

```
# If selecting a cell with atleast 1 mine in neighboring cells
```

```
else:
    mine_values[r][col] = numbers[r][col]
```

End game

There is a requirement to check for completion of the game, each time a move is made. This is done by:

```
# Check for game completion
if(check_over()):
    show_mines()
    print_mines_layout()
    print("Congratulations!!! YOU WIN")
    over = True
    continue
```

The function check_over(), is responsible for checking the completion of the game.

```
# Function to check for completion of the game
def check_over():
    global mine_values
    global n
    global mines_no

    # Count of all numbered values
    count = 0

    # Loop for checking each cell in the grid
    for r in range(n):
        for col in range(n):

            # If cell not empty or flagged
            if mine_values[r][col] != ' ' and mine_values[r][col] != 'F':
                count = count + 1
    # Count comparison
    if count == n * n - mines_no:
        return True
    else:
        return False
```

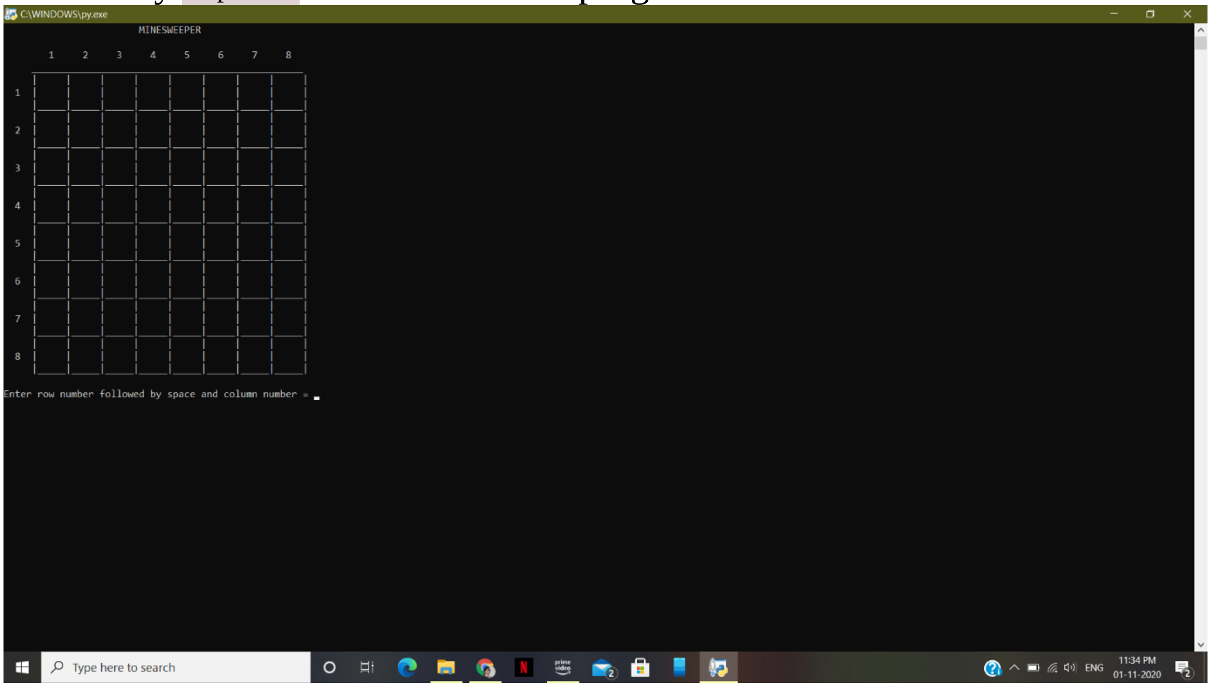
We count the number of cells, that are not empty or flagged. When this count is equal to the total cells, except those containing mines, then the game is regarded as over.

Clearing output after each move

The terminal becomes crowded as we keep on printing stuff on it. Therefore, there must be provision for clearing it constantly. This can be done by:

```
# Function for clearing the terminal
def clear():
    os.system("clear")
```

Note: There is a need to import the os library, before using this feature. It can be done by 'import os' at the start of the program.



```
                    MINESWEEPER

         1     2     3     4     5     6     7     8
      ________________________________________________
      |     |     |     |     |     |     |     |     |
  1   |     |     |     |     |     |     |     |     |
      |_____|_____|_____|_____|_____|_____|_____|_____|
      |     |     |     |     |     |     |     |     |
  2   |     |     |     |     |     |     |     |     |
      |_____|_____|_____|_____|_____|_____|_____|_____|
      |     |     |     |     |     |     |     |     |
  3   |     |     |     |     |     |     |     |     |
      |_____|_____|_____|_____|_____|_____|_____|_____|
      |     |     |     |     |     |     |     |     |
  4   |     |     |     |     |     |     |     |     |
      |_____|_____|_____|_____|_____|_____|_____|_____|
      |     |     |     |     |     |     |     |     |
  5   |     |     |     |     |     |     |     |     |
      |_____|_____|_____|_____|_____|_____|_____|_____|
      |     |     |     |     |     |     |     |     |
  6   |     |     |     |     |     |     |     |     |
      |_____|_____|_____|_____|_____|_____|_____|_____|
      |     |     |     |     |     |     |     |     |
  7   |     |     |     |     |     |     |     |     |
      |_____|_____|_____|_____|_____|_____|_____|_____|
      |     |     |     |     |     |     |     |     |
  8   |     |     |     |     |     |     |     |     |
      |_____|_____|_____|_____|_____|_____|_____|_____|

Enter row number followed by space and column number = 
```

Conclusion

We hope that this tutorial on creating our own Minesweeper game was understandable as well as fun. For any queries, feel free to comment below. The complete code is also available on https://github.com/SBharathbr/Minesweeper.git

Reference

- Adamatzky, Andrew (1997). "How cellular automaton plays Minesweeper". Applied Mathematics and Computation.
- Lakshtanov, Evgeny; Oleg German (2010). "'Minesweeper' and spectrum of discrete Laplacians". Applicable Analysis. .
- Kaye, Richard (2000). "Minesweeper is NP-complete". Mathematical Intelligencer. Further information available online at Richard Kaye's Minesweeper pages.
- Mordechai Ben-Ari (2018). Minesweeper is NP-Complete (PDF) (Report). Weizmann Institute of Science, Department of Science Teaching. — An open-access paper explaining Kaye's NP-completeness result.