
*Figure 1*

# Foundation Certificate for Higher Education

Module: DOC334 Computer Programming
Module Leader: Mr. Nishan Saliya
Assignment Type: Individual
Submission Date: 2024.03.30

Student ID: 20231264
Student Name: Ranuga Disansa Belpa Gamage
Student Email: ranuga.20231264@iit.ac.lk

# Acknowledgment

I want to express my appreciation to those who have helped complete this report.

I have been able to complete this report because of the Academic advisors Mr. Nishan Saliya and Ms. Shafka Fuard for their invaluable support, mentorship, and feedback, and the faculty members of DOC 334 (Computer Programming).

Additionally, I would like to thank my family members for their unwilling encouragement and help throughout the report.

Thank you.

# Table of Contents

# Table of Figures

# 1. Introduction

## 1.1 The Problem

The objective is to develop a Python program to simulate the process of percolation, which is the process of liquid slowly passing through a filter. There need to be two outputs of a text file (.txt) and an HTML file (.html) which contains a grid containing 2 random numbers between 10 and 99 and with random empty cells. Percolation is achievable in an entire column in the grid without empty cells, percolation isn't achieved if the column has empty cells.

## 1.2 The Solution

The solution that is created is for the problem to simulate percolation where liquid flows through a filter, and the main objective of the simulation is to check if percolation is achievable or not. A grid containing random two digital numbers, with random empty cells. Percolation is achieved according to the condition that an entire column in the grid is without empty cells in turn, otherwise, percolation isn't achieved. The program will output 2 files, a text file (.txt) as well as an HTML File (.html) allowing one to visualize the percolation process easily. The solution is a simulation system built using Python programming language.

# 2. Algorithm

The solution which is implemented by Python Programming language is stated below in the form of algorithm steps and with an explanation of how each aspect of the program modules, classes, and functions.

## 2.1 Pseudocode

### 2.1.1 Precolation/__init__.py

```
1.  IMPORT sys
2.  IMPORT random
3.  IMPORT os 4. IMPORT datetime
5.
6.      TRY:
7.      FROM prettytable IMPORT PrettyTable
8.      EXCEPT:
9.      os.system("pip install prettytable") 10.     FROM prettytable IMPORT PrettyTable
11.     ELSE:
12.     os.system("python3 -m pip install prettytable")
13.     FROM prettytable IMPORT PrettyTable
14.     finally:
15.     os.system("pip3 install prettytable") 16.     FROM prettytable IMPORT PrettyTable
17.
18.
19. FROM Percolation.helper_functions IMPORT *
20. FROM Percolation.grid_maker IMPORT *
21. FROM Percolation.txt IMPORT *
22. FROM Percolation.html IMPORT *
```

### 2.1.2 Precolation/helper_functions.py

```
1. FROM Percolation IMPORT random, PrettyTable, os
2.
3.
4.          DEFINE CLASS Random:
5.          @staticmethod
6.          DEFINE FUNCTION generate_random_number(start: int, end: int) -> int:
7.          """
8.          Generate a random integer number between start and end (inclusive).
9.
10.             Args:
11.             start (int): The minimum value of the random integer.
12.             end (int): The maximum value of the random integer (inclusive).
```

```
13.
14.         Returns:
15.         int: A random integer number between start and end.
16.         """
17.         RETURN random.randint(
18.         start, end
19.         )  # Create a random integer number between start and end
20.
21.     @staticmethod
22.     DEFINE FUNCTION select_choice(choices: list, probability: tuple[float]) -> list:
23.     """
24.     Select a random choice FROM a list based on a given probability distribution.
25.
26.         Args:
27.         choices (list): A list of possible choices.
28.         probability (tuple[float]): A tuple of probabilities corresponding to each choice IN
            `choices`. The sum of all probabilities must be equal to 1.
29.
30.     Returns:
31.     list: A single randomly selected choice FROM `choices` based on the given probability
        distribution.
32.
33.             Raises:
34.             ValueError: If the length of `choices` and `probability` do not match.
35.             """
36.             IF len(choices) EQUALS len(probability):
37.             RETURN random.choices(
38.             choices, weights=probability
39.             )  # Return a random choice FROM the given probability distribution
40.             raise ValueError(
41.             f"The list `choices` and tuple `probability` must have the same length. {choices}
    ({len(choices)}) != {probability} ({len(probability)})"
42.             )  # Create a error IF the length of choices and length of probability is not equal
43.
44.
45.   DEFINE CLASS Ok_or_not:
46.   """
47.   This DEFINE CLASS takes a 2D list as INPUT and determines whether each column of the list
      contains all unique elements. 48.     """
49.
50.         DEFINE FUNCTION __init__(self, grid: list) -> None:
51.         """
52.         Args:
53.         grid (list): A 2D list of integers. Each row represents a column of the original list.
54.         """
55.         SET self.grid TO grid
```

```
56.         SET self.t TO PrettyTable(header=False)
57.         SET self.col_length TO len(self.grid[0])
58.         SET self.cols_data TO {
59.         i: [] FOR i IN range(self.col_length)
60.         }  # create a dictionary WITH column index id and WITH a value as a list
61.         SET self.filter TO lambda cols_data: [
62.         "NO" IF "" IN cols_data[col] else "OK" FOR col IN cols_data 63.          ]  # filter the
            columns to see IF there are empty cell IN a column
64.
65.     DEFINE FUNCTION generate(self) -> list:
66.     """
67.     This function generates a table showing whether each column of the INPUT list contains all
            unique elements.
68.     The output is a string containing the table IN HTML format.
69.
70.             Returns:
71.             str: The table showing whether each column of the INPUT list contains all unique
                elements.
72.             """
73.             FOR row IN self.grid:
74.             SET row TO row.copy()
75.             FOR ele IN row:
76.             SET idx TO row.index(ele)
77.             self.cols_data[idx].append(
78.             ele
79.             )  # Add each element to each column
80.             SET row[idx] TO None
81.             SET list_ok_or_not TO self.filter(self.cols_data)  # Filter all the columns
82.             self.t.add_row(list_ok_or_not)  # Add to the self.t table
83.             RETURN list_ok_or_not
84.
85.     DEFINE FUNCTION get_html(self) -> str:
86.     """
87.     This function RETURNs the table showing whether each column of the INPUT list contains
            all unique elements 88.          IN HTML format.
89.
90.     Returns:
91.     str: The table showing whether each column of the INPUT list contains all unique elements
            IN HTML format.
92.     """
93.     RETURN self.t.get_html_string(header=False)  # get a html output of the table
94.
95.     DEFINE FUNCTION get_string(self) -> str:
96.     """
97.     This function RETURNs the table showing whether each column of the INPUT list contains
            all unique elements 98.          IN string format.
```

```
99.
100.        Returns:
101.        str: The table showing whether each column of the INPUT list contains all unique
            elements IN string format.
102.        """
103.        RETURN self.t.get_string(header=False)  # get a string output of the table
104.
105.
106.    DEFINE FUNCTION grid_condition(dims: str) -> tuple:
107.    """
108.    This function takes a string INPUT IN the form of "x" separated row and column
            dimensions and RETURNs a tuple of integers representing the dimensions.
109.    The function ensures that the INPUT is IN the correct format and that the row and
            column dimensions are within the specified range.
110.    If the INPUT is not IN the correct format or the dimensions are outside the specified
            range, the function RETURNs a default value of (5, 5).
111.
112.        Args:
113.        dims (str): A string IN the form of "x" separated row and column dimensions.
114.
115.        Returns:
116.        tuple: A tuple of integers representing the row and column dimensions.
117.        """
118.        SET split TO dims.split(dims[int(len(dims)/2)]) # Splitting the dim by 'middle'
            character
119.        IF any([s.isnumeric() FOR s IN split]): # checking IF the picked character is a
            number
120.        SET split TO dims.split('x') # then try and split FROM 'x'
121.        IF len(split) EQUALS 2:  # Checking conditions
122.        SET rows, cols TO split
123.        IF rows.isnumeric() and cols.isnumeric(): 124.                    SET rows, cols TO
            list(map(int, split)) 125.              IF 3 <= rows <= 9 and 3 <= cols <= 9:
126.            RETURN rows, cols
127.            OUTPUT('The entered grid size is invalid… The default 5x5 Grid is used')
128.            RETURN 5, 5  # Returning the default grid sizes
129.
130.
131.    DEFINE FUNCTION director_creator(directory: str) -> bool:
132.    """
133.    Creates a directory WITH the given name IN the current working directory.
134.
135.        Parameters:
136.        directory (str): The name of the directory to create.
```

```
137.
138.            Returns:
139.            bool: True IF the directory was created, False IF the directory already exists.
140.            """
141.            IF not os.path.isdir(f"./{directory}"):  # checking IF the directory doesnt exist
142.            os.mkdir(f"./{directory}")  # making the directory
143.            RETURN True
144.            RETURN False
```

### 2.1.3 Precolation/grid_maker.py

```
1. FROM Percolation IMPORT Random, random, PrettyTable
2.
3.
4.      DEFINE CLASS Grid_Maker:
5.      SET def __init__(self, rows: int, cols: int, perc_of_empty: float TO 0.1) -> None:
6.      """
7.      Initialize a new instance of the Grid_Maker class.
8.
9.          Args:
10.         rows (int): The number of rows IN the grid.
11.         cols (int): The number of columns IN the grid.
12.         perc_of_empty (float, optional): The percentage of cells that should be left empty. Defaults
            to 0.1.
13.         """
14.         SET self.grid TO []
15.         SET self.rows TO rows
16.         SET self.cols TO cols
17.         SET self.tot TO rows * cols
18.         SET self.r TO Random()  # Creating a instance of Random() object
19.         SET self.perc_of_empty TO perc_of_empty
20.         SET self.coordinate_cols TO lambda x: [
21.         i[1] FOR i IN x
22.         ]  # getting the first index of `x` list
23.
24.     DEFINE FUNCTION __empty_cell(self, row: int, col: int) -> bool:
25.     """
26.     Checks IF the given row and column indices are within the bounds of the grid,
27.     and IF the cell at that location is not already empty. If both conditions are true, 28.      the
        method sets the cell value to an empty string and RETURNs True, indicating that
29.     the cell was successfully marked as empty. Otherwise, it RETURNs False.
30.
31.         Args:
32.         row (int): The row index of the cell to be checked.
33.         col (int): The column index of the cell to be checked.
```

```
34.
35.         Returns:
36.         bool: True IF the cell was successfully marked as empty, False otherwise.
37.         """
38.         IF (
39.         row
40.         <= self.rows  # Checking IF the row given is less than or equal to self.rows 41.             or
            col
42.             <= self.cols  # Checking IF the col given is less than or equal to self.cols
43.             and ""
44.             not IN [
45.             row[col] FOR row IN self.grid
46.             ]  # Make sure that column doesn't have empty cells 47.             and self.grid != []
48.         ):
49.         SET self.grid[row - 1][col - 1] TO ""
50.         RETURN True
51.         RETURN False
52.
53.     DEFINE FUNCTION __check_if_all_columns_have_empty_cell(self) -> bool:
54.     """
55.     This function checks IF any of the columns have empty space left to remove
56.
57.         Args:
58.         None
59.
60.     Returns:
61.     bool: A boolean checking the `cols_check` list and IF any of them has a True, True will be
        RETURNed other wise False would be RETURNed
62.     """
63.     SET cols_check TO [] 64.         FOR col IN range(self.cols):
65.             SET empty TO "" IN [
66.             row[col] FOR row IN self.grid
67.             ]
68.             cols_check.append(False IF empty else True) 69.         RETURN any(cols_check)
70.
71.     DEFINE FUNCTION generate_cells_to_empty(self) -> list:
72.     """
73.     This function generates a list of random row and column indices that represent
74.     the locations of the cells that should be left empty IN the grid. It does so by
75.     randomly selecting a row and column index, and checking IF the cell at that location
76.     is already empty or not. If the cell is empty, the function marks it as empty by
77.     setting its value to an empty string, and decrements the count of the remaining
78.     empty cells. The function continues to select random indices UNTIL the desired
79.     number of empty cells has been reached.
80.
81.         Args:
```

```
82.         None
83.
84.                 Returns:
85.                 list: A list of tuples, where each tuple represents the row and column indices
86.                 of a cell that should be left empty.
87.                 """
88.                 SET empty_no_of_cell TO int(self.tot * self.perc_of_empty)
89.                 SET empty_no_of_cell TO (
90.                 empty_no_of_cell - self.cols
91.                 IF empty_no_of_cell > self.cols
92.                 else self.cols - empty_no_of_cell
93.                 )  # Calculate the number of cells that has to be emptied
94.                 SET coordinates TO []
95.                 SET empty_cells_in_cols TO
                    self.__check_if_all_columns_have_empty_cell()
96.                 WHILE empty_no_of_cell != 0 and empty_cells_in_cols:
97.                 SET row, col TO random.randint(1, self.rows), random.randint(
98.                 1, self.cols
99.                 )  # Select a random row and column
100.                IF col not IN self.coordinate_cols(coordinates):
101.                SET deleted_status TO self.__empty_cell(
102.                row, col
103.                )  # getting the status of emptying a cell 104.                IF
                    deleted_status:
105.                empty_no_of_cell -= 1
106.                coordinates.append((row, col))  # add the coordinates
107.                SET empty_cells_in_cols TO
                    self.__check_if_all_columns_have_empty_cell()
108.                RETURN coordinates
109.
110.                DEFINE FUNCTION make_grid(self) -> list:
111.                """
112.                This function creates a grid of random numbers IN the range of 10 to 99.
113.                It iterates over the number of rows and columns IN the grid and appends a
                    random number to each cell.
114.                The function RETURNs the grid as a list of lists.
115.                """
116.                FOR _ IN range(self.rows):
117.                SET row TO []
118.                FOR _ IN range(self.cols):
119.                row.append(
120.                self.r.generate_random_number(10, 99)
121.                )  # add the random number generated to the list
122.                self.grid.append(row)  # add the random number list generated to the list
```

```
                    RETURN self.grid

            DEFINE FUNCTION create_table(self) -> PrettyTable:
            """
            This function creates a table using PrettyTable

    Args:
None

                Returns:
                PrettyTable: a table WITH all of the data IN the 2D grid `self.grid`
                """
                SET table TO PrettyTable(header=False) # Create a table            FOR row IN self.grid:
                table.add_row(row) # Add a row
                RETURN table
            DEFINE FUNCTION generate_string(
            self,          ) -> str:
              """
              This function generates a string representation of the grid.

                Returns:
                str: A string representation of the grid.
                """
                RETURN str(self.create_table().get_string())  # Return a string  without the header

              DEFINE FUNCTION generate_html(self):
              """
              This function generates an HTML representation of the grid

                Returns:
                str: An HTML representation of the grid.
                """
                RETURN str(self.create_table().get_html_string())  # Return the string without the header

              DEFINE FUNCTION grid_maker(self) -> list:
              self.make_grid()  # Make a grid
              self.generate_cells_to_empty()  # Empty random grid
              RETURN self.grid  # send the final grid
```

### 2.1.4 Precolation/txt.py

```
1. FROM Percolation IMPORT Grid_Maker, Ok_or_not, director_creator
2.
3.
4. DEFINE CLASS Text:
5.
6.      DEFINE FUNCTION __init__(self, grid: Grid_Maker, ok_or_not: Ok_or_not, file_name:
        str) -> None:
7.      """
8.      Initialize a new instance of the Text class.
9.
10.        Args:
11.        grid (Grid_Maker): an instance of the Grid_Maker DEFINE CLASS that generates the text
           grid
12.        ok_or_not (Ok_or_not): an instance of the Ok_or_not DEFINE CLASS that generates the
           "ok" or "not ok" message
13.        file_name (str): the name of the file to be created
14.        """
15.        SET self.grid TO grid
16.        SET self.ok_or_not TO ok_or_not
17.        SET self.file_name TO f"./{file_name}/{file_name}.txt" 18.
           director_creator(file_name)
19.
20.     DEFINE FUNCTION create_full_grid(self) -> str:
21.     """
22.     This function combines the output of the grid and the ok_or_not functions to create a full text
        grid.
23.
24.        Returns:
25.        str: the combined output of the grid and the ok_or_not functions
26.        """
27.        RETURN (
28.        self.grid.generate_string() + "\n" + self.ok_or_not.get_string() 29.        )  # create the
           string
30.
31.        DEFINE FUNCTION create_file(self) -> str:
32.        """
33.        This function creates a text file WITH the combined output of the grid and the ok_or_not
           functions.
34.        Returns:
35.        str: the full text grid WITH the "ok" or "not ok" message
36.
37.        """
38.        SET txt TO self.create_full_grid()  # getting the full string of the grid
39.        WITH open(self.file_name, "w") as f:
```

```
40.             f.write(txt)  # add to the file
41.             RETURN txt
```

### 2.1.5 Precolation/html.py

```
1. FROM Percolation IMPORT Grid_Maker, Ok_or_not, director_creator
2.
3.
4.          DEFINE CLASS HTML:
5.          DEFINE FUNCTION __init__(self, grid: Grid_Maker, ok_or_not: Ok_or_not, file_name:
            str) -> None:
6.          """
7.          Initialize the HTML class.
8.
9.              Args:
10.             grid (Grid_Maker): The grid maker object.
11.             ok_or_not (Ok_or_not): The ok or not object.
12.             file_name (str): The file name.
13.             """
14.             SET self.grid TO grid
15.             SET self.ok_or_not TO ok_or_not
16.             SET self.file_name TO f"./{file_name}/{file_name}.html"  # the directory
17.             # calling the director_creator() helper function 18.         director_creator(file_name)
19.
20.                 @staticmethod
21.                 DEFINE FUNCTION create_html_code(elements: str, name: str="Grid") -> str:
22.                 RETURN f"""
23.                 <!DOCTYPE html>
24.                 <html lang="en">
25.                 <head>
26.                 <title>{name}</title>
27.                 </head>
28.                 <body>{elements}</body>
29.                 </html>
30.                 """  # RETURN the default html code template
31.
32.         DEFINE FUNCTION create_full_grid(self) -> str:
33.         """
34.         This function generates the full HTML code FOR the Percolation simulation.
35.
36.             Returns:
37.             str: The full HTML code FOR the Percolation simulation.
38.             """
39.             RETURN (
40.             self.grid.generate_html() + self.ok_or_not.get_html()
41.             )  # RETURNing the combination of grid and ok or not
42.
```

```
43.     DEFINE FUNCTION create_file(self) -> str:
44.     """
45.     This function generates the full HTML code FOR the Percolation simulation.
46.
47.         Returns:
48.         str: The full HTML code FOR the Percolation simulation.
49.         """
50.         SET code TO self.create_html_code(
51.         self.create_full_grid()
52.         )  # generating the html code 53.         WITH open(self.file_name, "w") as f:
54.         f.write(code)  # inserting the code into the file
55.         RETURN code
```

### 2.1.6 main.py

```
1. FROM Percolation IMPORT grid_condition, Grid_Maker, Ok_or_not, HTML, Text, sys, datetime
2.
3.     IF __name__ EQUALS "__main__":
4.     SET rows, cols TO grid_condition(sys.argv[-1])
5.     SET gm TO Grid_Maker(rows, cols)
6.     SET two_dim_grid TO gm.grid_maker()
7.     SET oon TO Ok_or_not(two_dim_grid)
8.     SET ok_or_not_list TO oon.generate()
9.     SET file_name TO datetime.datetime.now().strftime("%Y_%m_%d_%H%M")
10.    HTML(gm, oon, file_name).create_file()
11.    SET t TO Text(gm, oon, file_name).create_file()
12.    OUTPUT(t)
```

## 2.2 Classes

### 2.2.1 Grid_Maker()

The class `Grid_Maker()` is designed to generate and manipulate grids for the use of simulating the percolation process. The class contains the following functions:

## 2.2.1.1 __init__()

```
def __init__(self, rows: int, cols: int, perc_of_empty: float = 0.1) -> None:
    """
    Initialize a new instance of the Grid_Maker class.

    Args:
        rows (int): The number of rows in the grid.
        cols (int): The number of columns in the grid.
        perc_of_empty (float, optional): The percentage of cells that should be left empty. Defaults to 0.1.
    """
    self.grid = []
    self.rows = rows
    self.cols = cols
    self.tot = rows * cols
    self.r = Random()  # Creating a instance of Random() object
    self.perc_of_empty = perc_of_empty
    self.coordinate_cols = lambda x: [
        i[1] for i in x
    ]  # getting the first index of `x` list
```

*Figure 1*

This function is the initialization function which runs when an instance of the class is created. The function takes in the parameters of `rows` which is the no. of rows expected from the grid, `cols` which is the columns expected from the grid, and `perc_of_empty` which is the percentage of cells that should be empty, and it is default at 0.1 (which means 10%). The function returns Nothing. The data gathered is added to the object attributes and self.grid which is an empty list that will contain the 2D array later on, self.r an instance of the Random() class,self.tot which finds the total number of elements in the grid, self.coordinate_cols which is a lambda function which takes the first element of the list `x` passed into the function those attributes are also added.

## 2.2.1.2 __empty_cell ()

```python
def __empty_cell(self, row: int, col: int) -> bool:
    """
    Checks if the given row and column indices are within the bounds of the grid,
    and if the cell at that location is not already empty. If both conditions are true,
    the method sets the cell value to an empty string and returns True, indicating that
    the cell was successfully marked as empty. Otherwise, it returns False.

    Args:
        row (int): The row index of the cell to be checked.
        col (int): The column index of the cell to be checked.

    Returns:
        bool: True if the cell was successfully marked as empty, False otherwise.
    """
    if (
        row
        <= self.rows  # Checking if the row given is less than or equal to self.rows
        or col
        <= self.cols  # Checking if the col given is less than or equal to self.cols
        and ""
        not in [
            row[col] for row in self.grid
        ]  # Make sure that column doesn't have empty cells
        and self.grid != []
    ):
        self.grid[row - 1][col - 1] = ""
        return True
    return False
```

*Figure 2*

Which is a private function (hidden function) that is used to empty a cell after the grid is generated. The function takes in 2 parameters `row` and `col` which both are of the data type `int` and the function returns a Boolean which if `True` means that the cell has been emptied, and `False` means that the cell hasn't been emptied due to the conditions not matching up. The condition for emptying a cell is: The `self.rows` has to be bigger or equal to the `row` variable, `self.cols` has to be bigger or equal to `col`, "" can't be in the column previously, this is to check the assumption that a column can only have 1 empty cell and self.grid can't be an empty list this is to make sure that the make_grid() function has been already executed and the self.grid function isn't empty. If the conditions are accepted the cell is emptied or "" is set as the value.

### 2.2.1.3 __check_if_all_columns_have_empty_cell()

```python
def __check_if_all_columns_have_empty_cell(self) -> bool:
    """
    This function checks if any of the columns have empty space left to remove

    Args:
        None

    Returns:
        bool: A boolean checking the `cols_check` list and if any of them has a True, True will be returned other wise False would be returned
    """
    cols_check = []
    for col in range(self.cols):
        empty = "" in [
            row[col] for row in self.grid
        ]
        cols_check.append(False if empty else True)
    return any(cols_check)
```

*Figure 3*

The function checks if there is a space in any of the columns to remove. First, a list is initiated which is used to add Boolean data type and at the end, we check if any of the added bools are True if they are True is returned otherwise False is returned. The function iterates through all of the `self.cols` and checks whether there is space if it is False is added to the list otherwise True is added to the list and at the end the list is checked and the condition is returned.

## 2.2.1.4 generate_cells_to_empty()

```python
def generate_cells_to_empty(self) -> list:
    """
    This function generates a list of random row and column indices that represent
    the locations of the cells that should be left empty in the grid. It does so by
    randomly selecting a row and column index, and checking if the cell at that location
    is already empty or not. If the cell is empty, the function marks it as empty by
    setting its value to an empty string, and decrements the count of the remaining
    empty cells. The function continues to select random indices until the desired
    number of empty cells has been reached.

    Args:
        None

    Returns:
        list: A list of tuples, where each tuple represents the row and column indices
        of a cell that should be left empty.
    """
    empty_no_of_cell = int(self.tot * self.perc_of_empty)
    empty_no_of_cell = (
        empty_no_of_cell - self.cols
        if empty_no_of_cell > self.cols
        else self.cols - empty_no_of_cell
    )  # Calculate the number of cells that has to be emptied
    coordinates = []
    empty_cells_in_cols = self.__check_if_all_columns_have_empty_cell()
    while empty_no_of_cell != 0 and empty_cells_in_cols:
        row, col = random.randint(1, self.rows), random.randint(
            1, self.cols
        )  # Select a random row and column
        if col not in self.coordinate_cols(coordinates):
            deleted_status = self.__empty_cell(
                row, col
            )  # getting the status of emptying a cell
            if deleted_status:
                empty_no_of_cell -= 1
                coordinates.append((row, col))  # add the coordinates
        empty_cells_in_cols = self.__check_if_all_columns_have_empty_cell()
    return coordinates
```

*Figure 4*

This function has no parameters, and it returns a list that contains all the coordinates that have been emptied, which aren't being used but are added for future improvements of the project. The function first generates the number of empty cells that should be in the grid using the `self.prec_of_empty`and `self.tot`, then it is re-calculated by checking if the empty_no_of_cells is higher than the number of cols then the empty_no_of_cells is subtracted by the `self. cols` variable otherwise `self.cols` is subtracted by empty_no_of_cells. Then the coordinates list is created and we use the ` __check_if_all_columns_have_empty_cell()` function to make sure that there are columns that have no empty spaces and if both conditions of `empty_no_of_cell` aren't 0 and there are columns without empty cells a while loop is started which first creates row and col which are random numbers and then we check if an element from the col hasn't been removed beforehand using the self.coordinates_cols lambda function and if the condition is true then the row and column are passed through to the self.__empty_cell() function and according to the response if the cell was deleted successfully then the empty_no_of_cell is subtracted and then

the coordinates (in a tuple) are added to the coordinates list. Then the `empty_cells_in_cols` variable is refreshed and the next iteration will start. Finally, the coordinates of the removed cells will be returned.

### 2.2.1.5 make_grid()

```
def make_grid(self) -> list:
    """
    This function creates a grid of random numbers in the range of 10 to 99.
    It iterates over the number of rows and columns in the grid and appends a random number to each cell.
    The function returns the grid as a list of lists.
    """
    for _ in range(self.rows):
        row = []
        for _ in range(self.cols):
            row.append(
                self.r.generate_random_number(10, 99)
            )  # add the random number generated to the list
        self.grid.append(row)  # add the random number list generated to the list
    return self.grid
```

*Figure 5*

This function creates the elements of the 2D grid using 2 for loops and using the `self.r` Random class initiation and by using its `generate_random_number()` function. Then each row is added to the self. grid list and the grid is returned as well.

### 2.2.1.6 create_table()

```
def create_table(self) -> PrettyTable:
    """
    This function creates a table using PrettyTable

    Args:
        None

    Returns:
        PrettyTable: a table with all of the data in the 2D grid `self.grid`
    """
    table = PrettyTable(header=False) # Create a table
    for row in self.grid:
        table.add_row(row) # Add a row
    return table
```

*Figure 6*

The function first initiates a PrettyTable instance and then adds all of the rows in the self. grid 2D list. Then finally it returns to the table created.

### 2.2.1.7 generate_string()

```
def generate_string(
    self,
) -> str:
    """
    This function generates a string representation of the grid.

    Returns:
        str: A string representation of the grid.
    """
    return str(self.create_table().get_string())  # Return a string  without the header
```

*Figure 7*

Create a table using the `self.create_table()` function and then use the `.get_string()` function on the returned PrettyTable instance. The function returns a string representation of the grid.

### 2.2.1.8 generate_html()

```
def generate_html(self):
    """
    This function generates an HTML representation of the grid

    Returns:
        str: An HTML representation of the grid.
    """
    return str(self.create_table().get_html_string())  # Return the string without the header
```

*Figure 8*

Create a table using the `self.create_table()` function and then use the `.get_html()` function on the returned PrettyTable instance. The function returns an HTML representation of the grid.

### 2.2.1.9 grid_maker()

```
def grid_maker(self) -> list:
    self.make_grid()  # Make a grid
    self.generate_cells_to_empty()  # Empty random grid
    return self.grid  # send the final grid
```

*Figure 9*

The function just calls the self.make_grid() function to create a grid of random numbers and then calls the self.generate_cells_to_empty() to make random numbers empty and then finally returns the self.grid 2D grid.

## 2.2.2 Random()

This function provides static methods to generate random numbers and select choices using a probability distribution. The function has static methods due to it being easier to manage and it is bundled together if further developments are made the Random() class can develop with it.

### 2.2.2.1 generate_random_number()

```python
@staticmethod
def generate_random_number(start: int, end: int) -> int:
    """
    Generate a random integer number between start and end (inclusive).

    Args:
        start (int): The minimum value of the random integer.
        end (int): The maximum value of the random integer (inclusive).

    Returns:
        int: A random integer number between start and end.
    """
    return random.randint(
        start, end
    )  # Create a random integer number between start and end
```

*Figure 10*

The function takes in 2 parameters `start` and `end` which are the lowest possible and highest possible numbers that can be randomly picked. Then the `random` library's `random. randint()` function generates a random number between the start and end, and finally, the random number is returned.

### 2.2.2.2 select_choice()

```python
@staticmethod
def select_choice(choices: list, probability: tuple[float]) -> list:
    """
    Select a random choice from a list based on a given probability distribution.

    Args:
        choices (list): A list of possible choices.
        probability (tuple[float]): A tuple of probabilities corresponding to each choice in `choices`. The sum of all probabilities must be equal to 1.

    Returns:
        list: A single randomly selected choice from `choices` based on the given probability distribution.

    Raises:
        ValueError: If the length of `choices` and `probability` do not match.
    """
    if len(choices) == len(probability):
        return random.choices(
            choices, weights=probability
        )  # Return a random choice from the given probability distribution
    raise ValueError(
        f"The list `choices` and tuple `probability` must have the same length. {choices} ({len(choices)}) != {probability} ({len(probability)})"
    )  # Create a error if the length of choices and length of probability is not equal
```

*Figure 11*

The function takes in a choice (list) and weights (tuple with float values) and we first check if their lengths are the same and if not then a ValueError is raised. If the lengths match up then the choices and weights are passed into the `random.choices()` function of the `random` library in Python.

### 2.2.3 Ok_or_not()

Takes in a 2D grid and produces an HTML and string which tells whether or not each column of the grid has achieved percolation or not.

### 2.2.3.1 __init__()

```
def __init__(self, grid: list) -> None:
    """
    Args:
        grid (list): A 2D list of integers. Each row represents a column of the original list.
    """
    self.grid = grid
    self.t = PrettyTable(header=False)
    self.col_length = len(self.grid[0])
    self.cols_data = {
        i: [] for i in range(self.col_length)
    }  # create a dictionary with column index id and with a value as a list
    self.filter = lambda cols_data: [
        "NO" if "" in cols_data[col] else "OK" for col in cols_data
    ]  # filter the columns to see if there are empty cell in a column
```

*Figure 12*

This function is initiated with the class itself, the function takes in the grid (2D) and that is set as a global attribute, and then an instance of the PrettyTable is added as well, and the length of the column is saved as well, create a dictionary with an index and a list for all of the cols, and a lambda function is established which checks a specific column and checks if there is an “” in the column if there is “NO” is added to the list if not “OK” is added to the list.

### 2.2.3.2 generate()

```
def generate(self) -> list:
    """
    This function generates a table showing whether each column of the input list contains all unique elements.
    The output is a string containing the table in HTML format.

    Returns:
        str: The table showing whether each column of the input list contains all unique elements.
    """
    for row in self.grid:
        for ele in row:
            self.cols_data[row.index(ele)].append(
                ele
            )  # Add each element to each column
    list_ok_or_not = self.filter(self.cols_data)  # Filter all the columns
    self.t.add_row(list_ok_or_not)  # Add to the self.t table
    return list_ok_or_not
```

*Figure 13*

*The function doesn’t have any parameters, it returns a list of whether the columns have achieved percolation or not. The function first goes through the entire grid using 2 loops and adds all of the elements into the self.cols_data dictionary and then use the self.filter() function on the self.cols_data and then add the list returned by the function to the pretty table instance.*

### 2.2.3.3 get_html()

```
def get_html(self) -> str:
    """
    This function returns the table showing whether each column of the input list contains all unique elements
    in HTML format.

    Returns:
        str: The table showing whether each column of the input list contains all unique elements in HTML format.
    """
    return self.t.get_html_string(header=False)  # get a html output of the table
```

*Figure 14*

`get_html()` function produces a html string using the `self.t` PrettyTable. It returns the string with the HTML table.

### 2.2.3.4 get_string()

```
def get_string(self) -> str:
    """
    This function returns the table showing whether each column of the input list contains all unique elements
    in string format.

    Returns:
        str: The table showing whether each column of the input list contains all unique elements in string format.
    """
    return self.t.get_string(header=False)  # get a string output of the table
```

*Figure 15*

`get_string()` function produces a string using the `self.t` PrettyTable and returns the string.

## 2.2.4 HTML()

This class is responsible for creating an HTML file of the percolation simulation, and at the end, the class produces an HTML file at a specified file location with the current datetime.

### 2.2.3.1 __init__()

```
def __init__(self, grid: Grid_Maker, ok_or_not: Ok_or_not, file_name: str) -> None:
    """
    Initialize the HTML class.

    Args:
        grid (Grid_Maker): The grid maker object.
        ok_or_not (Ok_or_not): The ok or not object.
        file_name (str): The file name.
    """
    self.grid = grid
    self.ok_or_not = ok_or_not
    self.file_name = f"./{file_name}/{file_name}.html"  # the directory
    # calling the director_creator() helper function
    director_creator(file_name)
```

*Figure 16*

This function initiates with the creation of an instance of the class as well. This function takes in the grid in the form of the Grid_Maker class instance ok_or_not in the form of the Ok_or_not

class instance, and the file_name as well. The `director_creator()` function is called to make sure that the directory with the file name already exists. Nothing is returned in the function.

### 2.2.3.2 create_html_code()

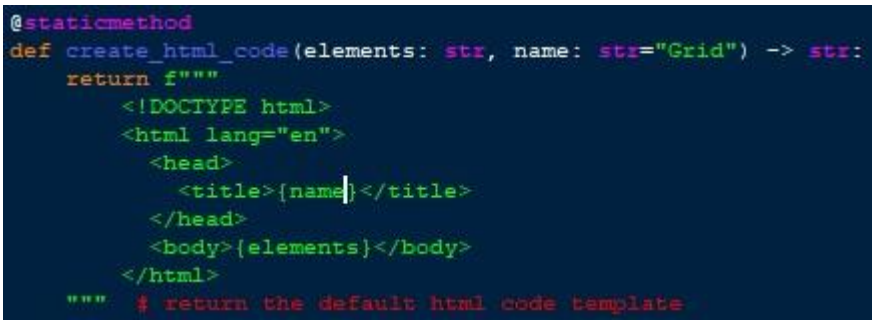


*Figure 17*

The `create_html_code()` contains a pre-defined string that has the basic structure of an HTML file, and it fills the structure with the parameters of `elements` and `name` which contain the body and title respectively. Finally, the function returns the structure with the updated body and title.

### 2.2.3.3 create_full_grid()

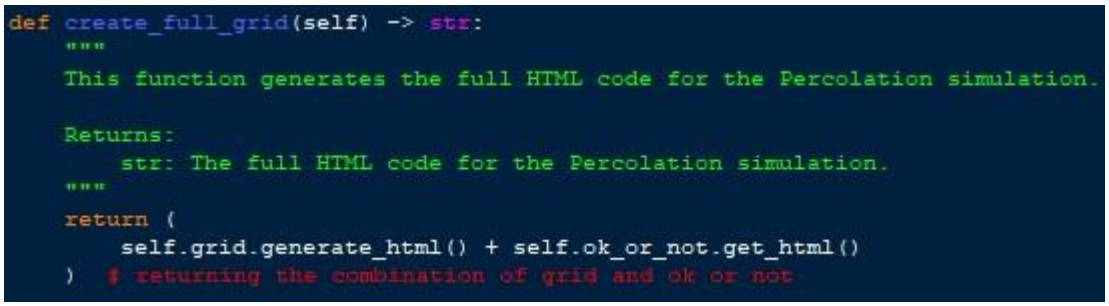


*Figure 18*

This function generates the entire HTML code by using the `Grid_Maker` and `Ok_or_not` instances.

### 2.2.3.4 create_file()

```
def create_file(self) -> str:
    """
    This function generates the full HTML code for the Percolation simulation.

    Returns:
        str: The full HTML code for the Percolation simulation.
    """
    code = self.create_html_code(
        self.create_full_grid()
    )  # generating the html code
    with open(self.file_name, "w") as f:
        f.write(code)  # inserting the code into the file
    return code
```

*Figure 19*

This function utilizes both other functions (`create_html_code` and `create_full_grid`) and creates the final code. The function first creates the full grid (using the `create_full_grid`) and then passes the return from the function into `create_html_code` and then finally writes the created html code into the file (self.file_name) and returns the code as well.

## 2.2.4 Text()

The `Text` class combines the grid and ok or not tables into the text format and inserts it into a file.

### 2.2.4.1 __init__()

```
def __init__(self, grid: Grid_Maker, ok_or_not: Ok_or_not, file_name: str) -> None:
    """
    Initialize a new instance of the Text class.

    Args:
        grid (Grid_Maker): an instance of the Grid_Maker class that generates the text grid
        ok_or_not (Ok_or_not): an instance of the Ok_or_not class that generates the "ok" or "not ok" message
        file_name (str): the name of the file to be created
    """
    self.grid = grid
    self.ok_or_not = ok_or_not
    self.file_name = f"./{file_name}/{file_name}.txt"
    director_creator(file_name)
```

*Figure 20*

This function runs when the Text class is initiated and it takes in 3 parameters which are: grid which is an instance of the Grid_Maker class, ok_or_not which is an instance of the Ok_or_not, and file_name which is the file name that the string should be saved in. This function runs the `director_creator()` function to make sure that a directory with the file name is created.

### 2.2.4.2 create_full_grid()

```
def create_full_grid(self) -> str:
    """
    This function combines the output of the grid and the ok_or_not functions to create a full text grid.

    Returns:
        str: the combined output of the grid and the ok_or_not functions
    """
    return (
        self.grid.generate_string() + "\n" + self.ok_or_not.get_string()
    )  # create the string
```

*Figure 21*

`create_full_grid()` function gets the string from both the Grid_Maker and Ok_or_not class instnacnes and combines them and returns them.

### 2.2.4.3 create_file()

```
def create_file(self) -> str:
    """
    This function creates a text file with the combined output of the grid and the ok_or_not functions.
    Returns:
        str: the full text grid with the "ok" or "not ok" message

    """
    txt = self.create_full_grid()  # getting the full string of the grid
    with open(self.file_name, "w") as f:
        f.write(txt)  # add to the file
    return txt
```

*Figure 22*

This function gets the combined string from `self.create_full_grid()` and then writes the returned string into the text file.

## 2.3 Functions

### 2.3.1 grid_condition()

```
def grid_condition(dims: str) -> tuple:
    """
    This function takes a string input in the form of "x" separated row and column dimensions and returns a tuple of integers representing the dimensions.
    The function ensures that the input is in the correct format and that the row and column dimensions are within the specified range.
    If the input is not in the correct format or the dimensions are outside the specified range, the function returns a default value of (5, 5).

    Args:
        dims (str): A string in the form of "x" separated row and column dimensions.

    Returns:
        tuple: A tuple of integers representing the row and column dimensions.
    """
    split = dims.split(dims[int(len(dims)/2)]) # Splitting the dim by 'middle' character
    if any([s.isnumeric() for s in split]): # checking if the picked character is a number
        split = dims.split('x') # then try and split from 'x'
    if len(split) == 2:  # Checking conditions
        rows, cols = split
        if rows.isnumeric() and cols.isnumeric():
            rows, cols = list(map(int, split))
            if 3 <= rows <= 9 and 3 <= cols <= 9:
                return rows, cols
    return 5, 5  # Returning the default grid sizes
```

*Figure 23*

This function is used to make sure that the entered grid size fits the criteria. We try and split the middle event so that in the case of the user entering a space instead of an `x` the program will still work, and in the worst-case scenario that it isn't the entered console argument is split by `x`,

and then we check for the length of the split list and then check if they match the conditions of both the rows and columns being bigger than or equal to 3 and less than or equal to 9. The parameter is dims which is the console argument from the user and a tuple is returned.

### 2.3.2 director_creator()

```
def director_creator(directory: str) -> bool:
    """
    Creates a directory with the given name in the current working directory.

    Parameters:
        directory (str): The name of the directory to create.

    Returns:
        bool: True if the directory was created, False if the directory already exists.
    """
    if not os.path.isdir(f"./{directory}"):  # checking if the directory doesnt exist
        os.mkdir(f"./{directory}")  # making the directory
        return True
    return False
```

*Figure 24*

The function `director_creator()` checks whether a directory already exists if not then it creates one and returns True, if it already exists it returns False. The parameter is the directory, and the return is a Boolean.

## 2.4 Packages Used

### 2.4.1 sys

The `sys` package is used in the Python program to get the console arguments from the user, and then it passes through to the `grid_condition()` function and finally, we get the rows and cols that the user entered.

### 2.4.2 random

The `random` package is used in many occasions throughout the program. It is used in the following situations: Finding a random coordinate in the grid to empty, Random().generate_random_number() to create a random number and random.randint() is used, Random().select_choice() to select a choice using probability where `random.choices()` is used.

### 2.4.3 os

`os` package is used in the program to make sure that the `prettytable` library is installed using `os.system()` and in the `director_creator()` helper_function where `os.path.isdir()` and `os.mkdir()` is used.

### 2.4.4 datetime

The `datetime` package is used to create the file name of the current time for the .html and .txt file.

### 2.4.5 prettytable

The `prettytable` package is one of the most important and useful packages used in the program, the prettytable is used to create a table with a structure it is extremely easy to use and flexible. The `prettytable` is used in Grid_Maker().create_table(), Ok_or_Not() initialization. All of the tables throughout the program use prettytable and its .get_string() and .get_html() are used for the filesaving functions.

## 2.5 Project Structure

The following is the project structure that has been used for this program.

```
Percolation/
│
├── Percolation/
│   ├── __init__.py
│   ├── Grid_Maker.py
│   ├──helper_functions.py
│   ├──html.py
│   ├──txt.py
│
└── main.py
```

### 2.5.1 Precolation/__init__.py

This file marks the directory as a Python module. This is like __init__ in a Class and the __init__.py file runs when we use Percolation.

### 2.5.2 Precolation/Grid_Maker.py

This file contains the Grid_Maker class used to generate grids.

### 2.5.3 Precolation/helper_function.py

`helper_function.py` contains multiple classes of `Random` which has all of the aspects of randomization, `Ok_or_not` class used to generate a table regarding whether the percolation of the columns in the grid is achieved or not, `grid_condition()` which makes sure that entered grid dimensions are acceptable and `director_creator()` used to create a directory if it doesn't already exist.

### 2.5.4 Precolation/html.py

This file contains the class `HTML` which is used to save the .html file of the grid and ok or not.

### 2.5.4 Precolation/txt.py

This file contains the class `Text` which is used to save the .txt file of the grid and ok or not.

### 2.5.4 main.py

This is the main file that combines all the modules and creates the grid.

## 2.6 Explanation

```
PS D:\University\IIT\Second Semester\DOC334-ICW> python3 .\main.py
The enetered grid size is invalid... The default 5x5 Grid is used
+----+----+----+----+----+
| 75 | 76 | 40 | 83 | 46 |
| 55 | 91 |    | 75 | 56 |
| 26 | 57 | 61 | 66 | 83 |
| 53 |    | 40 | 47 | 19 |
|    | 34 | 58 | 74 | 84 |
+----+----+----+----+----+
+----+----+----+----+----+
| NO | NO | NO | OK | OK |
+----+----+----+----+----+
```

*Figure 25*

When the main.py file runs the first takes the `sys.argv[-1]` and passes it through to the `grid_condition()` and the function will return the rows and cols, then the rows and cols are passed through to the `Grid_Maker()` and we execute the `Grid_Maker().grid_maker()` function which produces a two dimensional grid, then we pass the two dimentainal grid through to `Ok_or_not()`where we call the

`Ok_or_not().generate()` function to check if the columns in the grid (2D) has achieved percolation or not and it returns a list if percolation succeed or not for each column. Then we get the current datetime and then pass along the `Grid_Maker()` instance and `Ok_or_not()` instance and the `file_name` which is the currentdatetime to `HTML()` and call `.create_file()` which creates the .html file then we do the same process to the `Text()` class and call `.create_file()` then the Text File will be created. Finally, the table is printed as the output.

# 3 Assumptions

The listed below are assumptions that were made about the solution when making the program:

1. Each Column Can Have Only 1 Empty Cell
   It is assumed that every column in the grid can have at the most 1 empty cell, this assumption was made due to all the Grids in the Course Work Specification Containing only 1 Cell Maximum Per Column.

# 4 Python Code

## 4.1 Precolation/__init__.py

```
import sys
import random
import os
import datetime

try:
    from prettytable import PrettyTable
except:
    os.system("pip install prettytable")
    from prettytable import PrettyTable
else:
    os.system("python3 -m pip install prettytable")
    from prettytable import PrettyTable
finally:
    os.system("pip3 install prettytable")
    from prettytable import PrettyTable


from Percolation.helper_functions import *
from Percolation.grid_maker import *
from Percolation.txt import *
from Percolation.html import *
```

# 4.2 Precolation/grid_maker.py

```
from Percolation import Random, random, PrettyTable


class Grid_Maker:
    def __init__(self, rows: int, cols: int, perc_of_empty: float = 0.1) -> None:
        """
        Initialize a new instance of the Grid_Maker class.

        Args:
            rows (int): The number of rows in the grid.
            cols (int): The number of columns in the grid.
            perc_of_empty (float, optional): The percentage of cells that should be left empty.
Defaults to 0.1.
        """
        self.grid = []
        self.rows = rows
        self.cols = cols
        self.tot = rows * cols
        self.r = Random()  # Creating a instance of Random() object
        self.perc_of_empty = perc_of_empty
        self.coordinate_cols = lambda x: [
            i[1] for i in x
        ]  # getting the first index of `x` list

    def __empty_cell(self, row: int, col: int) -> bool:
        """
        Checks if the given row and column indices are within the bounds of the grid,
        and if the cell at that location is not already empty. If both conditions are true,
        the method sets the cell value to an empty string and returns True, indicating that
        the cell was successfully marked as empty. Otherwise, it returns False.

        Args:
            row (int): The row index of the cell to be checked.
            col (int): The column index of the cell to be checked.

        Returns:
            bool: True if the cell was successfully marked as empty, False otherwise.
        """
        if (
            row
            <= self.rows  # Checking if the row given is less than or equal to self.rows
            or col
            <= self.cols  # Checking if the col given is less than or equal to self.cols
            and ""
            not in [
```

```
                row[col] for row in self.grid
            ]  # Make sure that column doesn't have empty cells
            and self.grid != []
        ):
            self.grid[row - 1][col - 1] = ""
            return True
        return False

    def __check_if_all_columns_have_empty_cell(self) -> bool:
        """
        This function checks if any of the columns have empty space left to remove

        Args:
            None

        Returns:
            bool: A boolean checking the `cols_check` list and if any of them has a True, True
will be returned other wise False would be returned
        """
        cols_check = []
        for col in range(self.cols):
            empty = "" in [
                row[col] for row in self.grid
            ]
            cols_check.append(False if empty else True)
        return any(cols_check)

    def generate_cells_to_empty(self) -> list:
        """
        This function generates a list of random row and column indices that represent
        the locations of the cells that should be left empty in the grid. It does so by
        randomly selecting a row and column index, and checking if the cell at that location
        is already empty or not. If the cell is empty, the function marks it as empty by
        setting its value to an empty string, and decrements the count of the remaining
        empty cells. The function continues to select random indices until the desired
        number of empty cells has been reached.

        Args:
            None

        Returns:
            list: A list of tuples, where each tuple represents the row and column indices
            of a cell that should be left empty.
        """
        empty_no_of_cell = int(self.tot * self.perc_of_empty)
        empty_no_of_cell = (
            empty_no_of_cell - self.cols
```

```
            if empty_no_of_cell > self.cols
            else self.cols - empty_no_of_cell
        )  # Calculate the number of cells that has to be emptied
        coordinates = []
        empty_cells_in_cols = self.__check_if_all_columns_have_empty_cell()
        while empty_no_of_cell != 0 and empty_cells_in_cols:
            row, col = random.randint(1, self.rows), random.randint(
                1, self.cols
            )  # Select a random row and column
            if col not in self.coordinate_cols(coordinates):
                deleted_status = self.__empty_cell(
                    row, col
                )  # getting the status of emptying a cell
                if deleted_status:
                    empty_no_of_cell -= 1
            coordinates.append((row, col))  # add the coordinates
            empty_cells_in_cols = self.__check_if_all_columns_have_empty_cell()
        return coordinates

    def make_grid(self) -> list:
        """
        This function creates a grid of random numbers in the range of 10 to 99.
        It iterates over the number of rows and columns in the grid and appends a random
number to each cell.
        The function returns the grid as a list of lists.
        """
        for _ in range(self.rows):
            row = []
            for _ in range(self.cols):
                row.append(
                    self.r.generate_random_number(10, 99)
                )  # add the random number generated to the list
            self.grid.append(row)  # add the random number list generated to the list
        return self.grid

    def create_table(self) -> PrettyTable:
        """
        This function creates a table using PrettyTable

        Args:
            None

        Returns:
            PrettyTable: a table with all of the data in the 2D grid `self.grid`
        """
        table = PrettyTable(header=False) # Create a table
        for row in self.grid:
```

```
            table.add_row(row) # Add a row
        return table

    def generate_string(
        self,
    ) -> str:
        """
        This function generates a string representation of the grid.

        Returns:
            str: A string representation of the grid.
        """
        return str(self.create_table().get_string())  # Return a string  without the header

    def generate_html(self):
        """
        This function generates an HTML representation of the grid

        Returns:
            str: An HTML representation of the grid.
        """
        return str(self.create_table().get_html_string())  # Return the string without the header

    def grid_maker(self) -> list:
        self.make_grid()  # Make a grid
        self.generate_cells_to_empty()  # Empty random grid
        return self.grid  # send the final grid
```

## 4.3 Precolation/helper_functions.py

```
from Percolation import random, PrettyTable, os


class Random:
    @staticmethod
    def generate_random_number(start: int, end: int) -> int:
        """
        Generate a random integer number between start and end (inclusive).

        Args:
            start (int): The minimum value of the random integer.
            end (int): The maximum value of the random integer (inclusive).

        Returns:
            int: A random integer number between start and end.
        """
        return random.randint(
            start, end
```

```
        )  # Create a random integer number between start and end

    @staticmethod
    def select_choice(choices: list, probability: tuple[float]) -> list:
        """
        Select a random choice from a list based on a given probability distribution.

        Args:
            choices (list): A list of possible choices.
            probability (tuple[float]): A tuple of probabilities corresponding to each choice in
`choices`. The sum of all probabilities must be equal to 1.

        Returns:
            list: A single randomly selected choice from `choices` based on the given
probability distribution.

        Raises:
            ValueError: If the length of `choices` and `probability` do not match.
        """
        if len(choices) == len(probability):
            return random.choices(
                choices, weights=probability
            )  # Return a random choice from the given probability distribution
        raise ValueError(
            f"The list `choices` and tuple `probability` must have the same length. {choices}
({len(choices)}) != {probability} ({len(probability)})"
        )  # Create a error if the length of choices and length of probability is not equal


class Ok_or_not:
    """
    This class takes a 2D list as input and determines whether each column of the list
contains all unique elements.
    """

    def __init__(self, grid: list) -> None:
        """
        Args:
            grid (list): A 2D list of integers. Each row represents a column of the original list.
        """
        self.grid = grid
        self.t = PrettyTable(header=False)
        self.col_length = len(self.grid[0])
        self.cols_data = {
            i: [] for i in range(self.col_length)
        }  # create a dictionary with column index id and with a value as a list
        self.filter = lambda cols_data: [
```

```
            "NO" if "" in cols_data[col] else "OK" for col in cols_data
        ]  # filter the columns to see if there are empty cell in a column

    def generate(self) -> list:
        """
        This function generates a table showing whether each column of the input list contains
all unique elements.
        The output is a string containing the table in HTML format.

        Returns:
            str: The table showing whether each column of the input list contains all unique
elements.
        """
        for row in self.grid:
            row = row.copy()
            for ele in row:
                idx = row.index(ele)
                self.cols_data[idx].append(
                    ele
                )  # Add each element to each column
                row[idx] = None
        list_ok_or_not = self.filter(self.cols_data)  # Filter all the columns
        self.t.add_row(list_ok_or_not)  # Add to the self.t table
        return list_ok_or_not

    def get_html(self) -> str:
        """
        This function returns the table showing whether each column of the input list contains
all unique elements
        in HTML format.

        Returns:
            str: The table showing whether each column of the input list contains all unique
elements in HTML format.
        """
        return self.t.get_html_string(header=False)  # get a html output of the table

    def get_string(self) -> str:
        """
        This function returns the table showing whether each column of the input list contains
all unique elements
        in string format.

        Returns:
            str: The table showing whether each column of the input list contains all unique
elements in string format.
        """
```

```
        return self.t.get_string(header=False)  # get a string output of the table


def grid_condition(dims: str) -> tuple:
    """
    This function takes a string input in the form of "x" separated row and column
dimensions and returns a tuple of integers representing the dimensions.
    The function ensures that the input is in the correct format and that the row and column
dimensions are within the specified range.
    If the input is not in the correct format or the dimensions are outside the specified range,
the function returns a default value of (5, 5).

    Args:
        dims (str): A string in the form of "x" separated row and column dimensions.

    Returns:
        tuple: A tuple of integers representing the row and column dimensions.
    """
    split = dims.split(dims[int(len(dims)/2)]) # Splitting the dim by 'middle' character
    if any([s.isnumeric() for s in split]): # checking if the picked character is a number
        split = dims.split('x') # then try and split from 'x'
    if len(split) == 2:  # Checking conditions
        rows, cols = split
        if rows.isnumeric() and cols.isnumeric():
            rows, cols = list(map(int, split))
            if 3 <= rows <= 9 and 3 <= cols <= 9:
                return rows, cols
    print('The enetered grid size is invalid... The default 5x5 Grid is used')
    return 5, 5  # Returning the default grid sizes


def director_creator(directory: str) -> bool:
    """
    Creates a directory with the given name in the current working directory.

    Parameters:
        directory (str): The name of the directory to create.

    Returns:
        bool: True if the directory was created, False if the directory already exists.
    """
    if not os.path.isdir(f"./{directory}"):  # checking if the directory doesnt exist
        os.mkdir(f"./{directory}")  # making the directory
        return True
    return False
```

# 4.4 Precolation/html.py

```
from Percolation import Grid_Maker, Ok_or_not, director_creator


class HTML:
    def __init__(self, grid: Grid_Maker, ok_or_not: Ok_or_not, file_name: str) -> None:
        """
        Initialize the HTML class.

        Args:
            grid (Grid_Maker): The grid maker object.
            ok_or_not (Ok_or_not): The ok or not object.
            file_name (str): The file name.
        """
        self.grid = grid
        self.ok_or_not = ok_or_not
        self.file_name = f"./{file_name}/{file_name}.html"  # the directory
        # calling the director_creator() helper function
        director_creator(file_name)

    @staticmethod
    def create_html_code(elements: str, name: str="Grid") -> str:
        return f"""
            <!DOCTYPE html>
            <html lang="en">
              <head>
                <title>{name}</title>
              </head>
              <body>{elements}</body>
            </html>
        """  # return the default html code template

    def create_full_grid(self) -> str:
        """
        This function generates the full HTML code for the Percolation simulation.

        Returns:
            str: The full HTML code for the Percolation simulation.
        """
        return (
            self.grid.generate_html() + self.ok_or_not.get_html()
        )  # returning the combination of grid and ok or not

    def create_file(self) -> str:
        """
        This function generates the full HTML code for the Percolation simulation.
```

```
        Returns:
            str: The full HTML code for the Percolation simulation.
        """
        code = self.create_html_code(
            self.create_full_grid()
        )  # generating the html code
        with open(self.file_name, "w") as f:
            f.write(code)  # inserting the code into the file
        return code
```

# 4.5 Precolation/txt.py

```
from Percolation import Grid_Maker, Ok_or_not, director_creator


class Text:

    def __init__(self, grid: Grid_Maker, ok_or_not: Ok_or_not, file_name: str) -> None:
        """
        Initialize a new instance of the Text class.

        Args:
            grid (Grid_Maker): an instance of the Grid_Maker class that generates the text grid
            ok_or_not (Ok_or_not): an instance of the Ok_or_not class that generates the "ok" or "not ok" message
            file_name (str): the name of the file to be created
        """
        self.grid = grid
        self.ok_or_not = ok_or_not
        self.file_name = f"./{file_name}/{file_name}.txt"
        director_creator(file_name)

    def create_full_grid(self) -> str:
        """
        This function combines the output of the grid and the ok_or_not functions to create a full text grid.

        Returns:
            str: the combined output of the grid and the ok_or_not functions
        """
        return (
            self.grid.generate_string() + "\n" + self.ok_or_not.get_string()
        )  # create the string

    def create_file(self) -> str:
        """
```

```
        This function creates a text file with the combined output of the grid and the
ok_or_not functions.
        Returns:
            str: the full text grid with the "ok" or "not ok" message

        """
        txt = self.create_full_grid()  # getting the full string of the grid
        with open(self.file_name, "w") as f:
            f.write(txt)  # add to the file
        return txt
```

## 4.6 main.py

```
from Percolation import grid_condition, Grid_Maker, Ok_or_not, HTML, Text, sys,
datetime

if __name__ == "__main__":
    rows, cols = grid_condition(sys.argv[-1])
    gm = Grid_Maker(rows, cols)
    two_dim_grid = gm.grid_maker()
    oon = Ok_or_not(two_dim_grid)
    ok_or_not_list = oon.generate()
    file_name = datetime.datetime.now().strftime("%Y_%m_%d_%H%M")
    HTML(gm, oon, file_name).create_file()
    t = Text(gm, oon, file_name).create_file()
    print(t)
```

# 5 Test Cases

## 5.1 Valid Grid

The following test case is the valid grid.

| Command | Input Entered | Expected Outcome | Actual Outcome | Results |
|---|---|---|---|---|
| python3 main.py 4x4 | 4x4 | Display a 4x4 Grid | Display a 4x4 Grid | Pass |

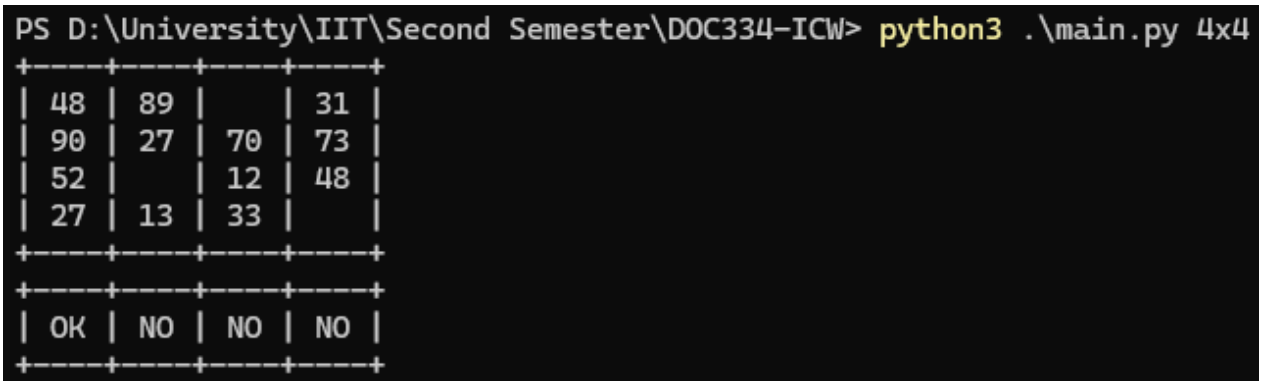


*Figure 26*

# 5.2 No Command Line Arguments (Default)

The following test case is the default grid.

| Command | Input Entered | Expected Outcome | Actual Outcome | Results |
|---|---|---|---|---|
| python3 main.py | | Display a Default 5x5 Grid | Display a Default 5x5 Grid | Pass |

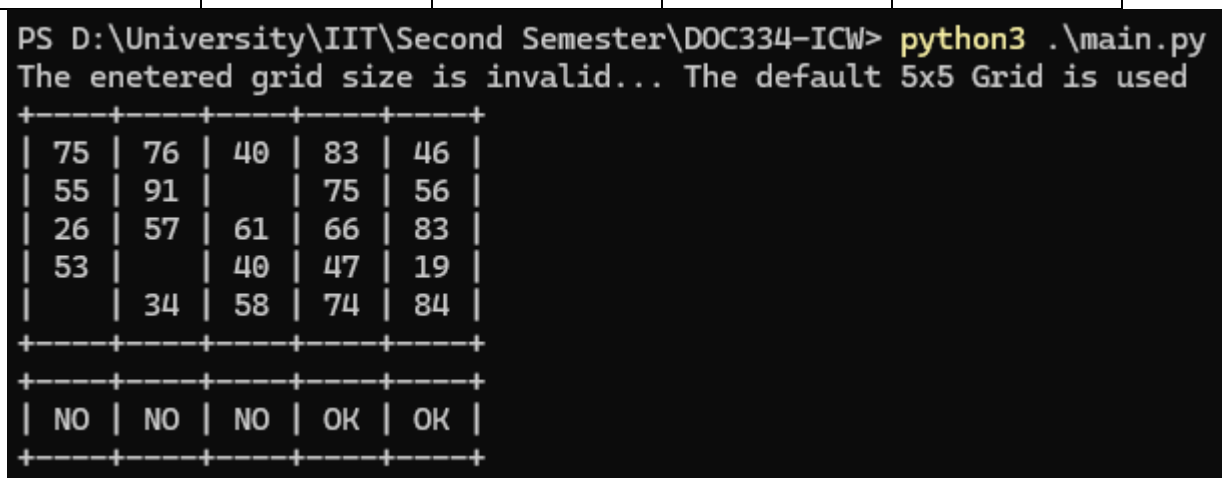


*Figure 27*

# 5.3 Invalid Grid

The following test case is the output for a invalid grid size, an error message and the use of the default 5x5 Grid.

| Command | Input Entered | Expected Outcome | Actual Outcome | Results |
|---|---|---|---|---|

| | | | | |
|---|---|---|---|---|
| python3 main.py 10x3 | 10x3 | Display a error message and Create a Default 5x5 Grid | "The enetered grid size is invalid... The default 5x5 Grid is used" & Create a 5X5 Grid | Pass |

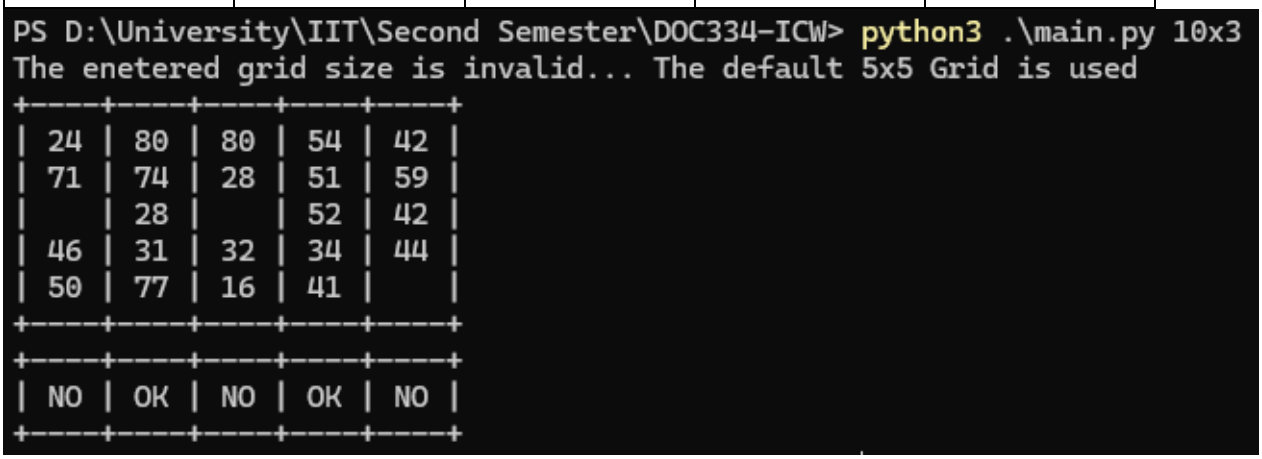


*Figure 28*

# 5.4 Impossible Percolation

The following test case is the output for impossible percolation, where each column will have a empty cell and the percolation status for each column is "NO".

| Command | Input Entered | Expected Outcome | Actual Outcome | Results |
|---|---|---|---|---|
| python3 main.py 3x3 | 3x3 | Display a 3x3 Grid with a Precolation Status of all "NO" | Display a 3x3 Grid with a Precolation Status of all "NO" | Pass |

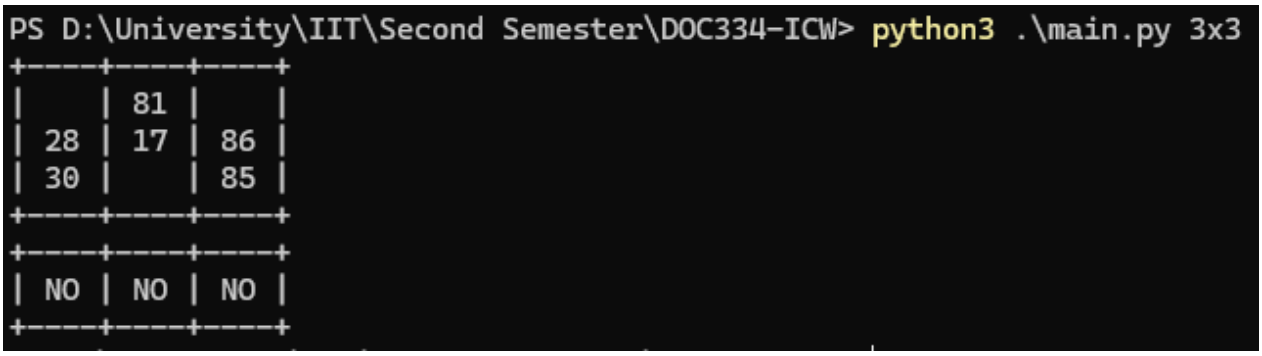


*Figure 29*

# 5.5 Minimum Grid Dimensions

The minimum grid size that percolation is simulated of.

| Command | Input Entered | Expected Outcome | Actual Outcome | Results |
|---|---|---|---|---|
| python3 main.py 3x3 | 3x3 | Display a 3x3 Grid | Display a 3x3 Grid | Pass |

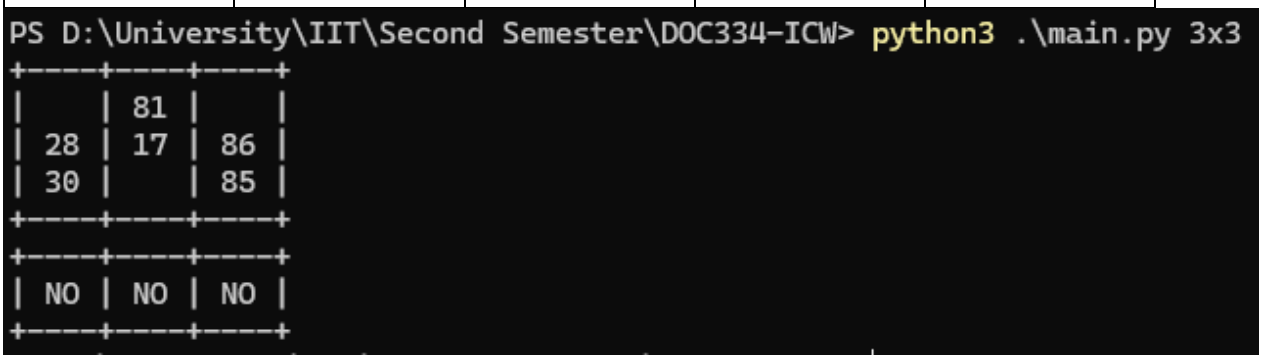


*Figure 30*

# 5.6 Maximum Grid Dimensions

The maximum grid size that percolation is simulated of.

| Command | Input Entered | Expected Outcome | Actual Outcome | Results |
|---|---|---|---|---|
| python3 main.py 9x9 | 9x9 | Display a 9x9 Grid | Display a 9x9 Grid | Pass |

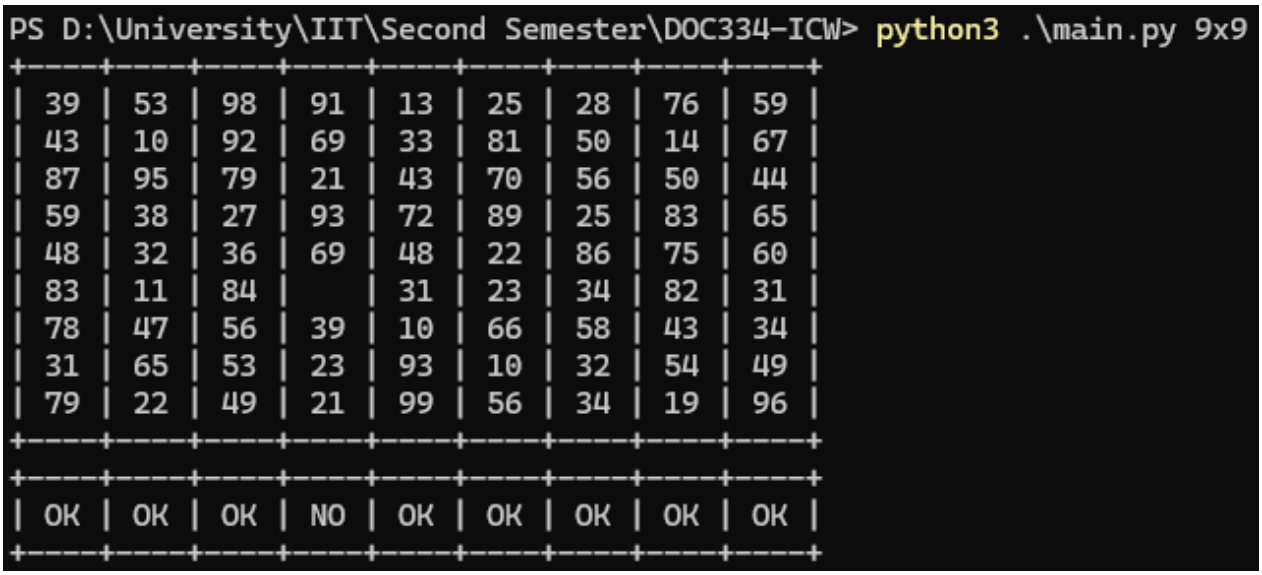


*Figure 31*

# 5.7 Text File Generation

A Text File being created.

| Command | Input Entered | Expected Outcome | Actual Outcome | Results |
|---|---|---|---|---|
| python3 main.py 6x5 | 6x5 | Display a 6x5 Grid and Create a text file with the 6x5 | Display a 6x5 Grid and Create a text file with the 6x5 | Pass |

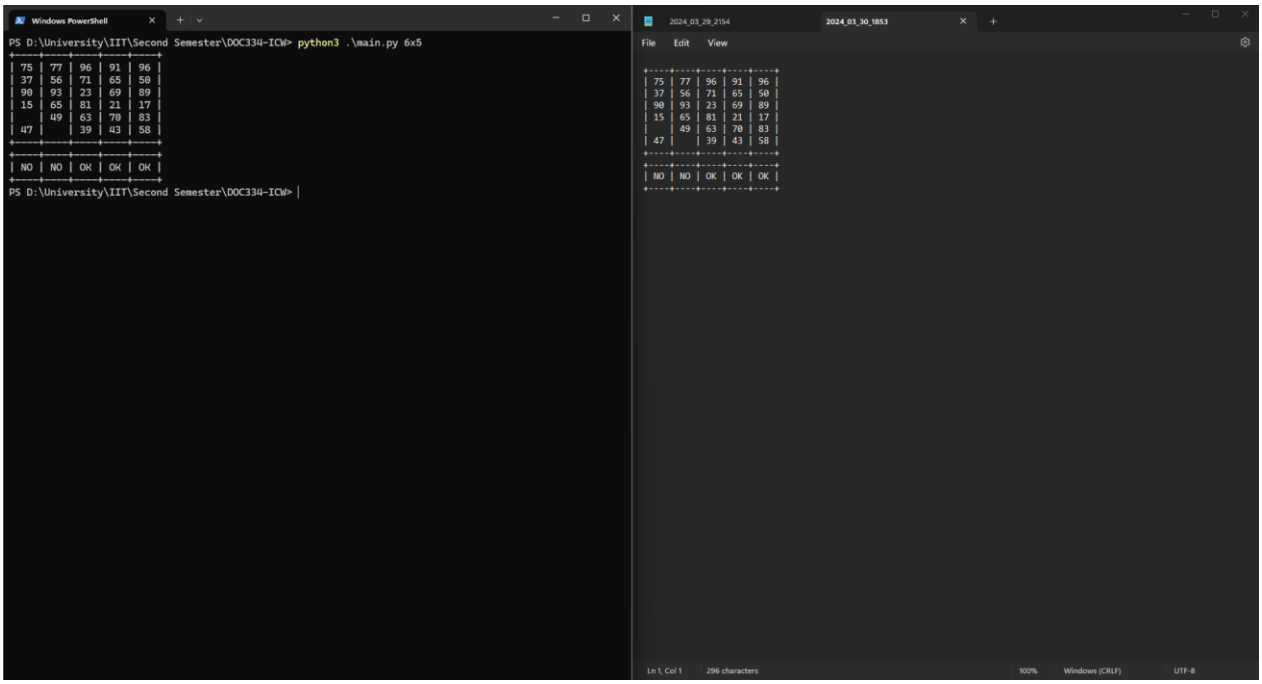


*Figure 32*

# 5.8 HTML File Generation

A HTML File being created.

| Command | Input Entered | Expected Outcome | Actual Outcome | Results |
|---|---|---|---|---|
| python3 main.py 6x5 | 6x5 | Display a 6x5 Grid and Create a html file with the 6x5 | Display a 6x5 Grid and Create a html file with the 6x5 | Pass |

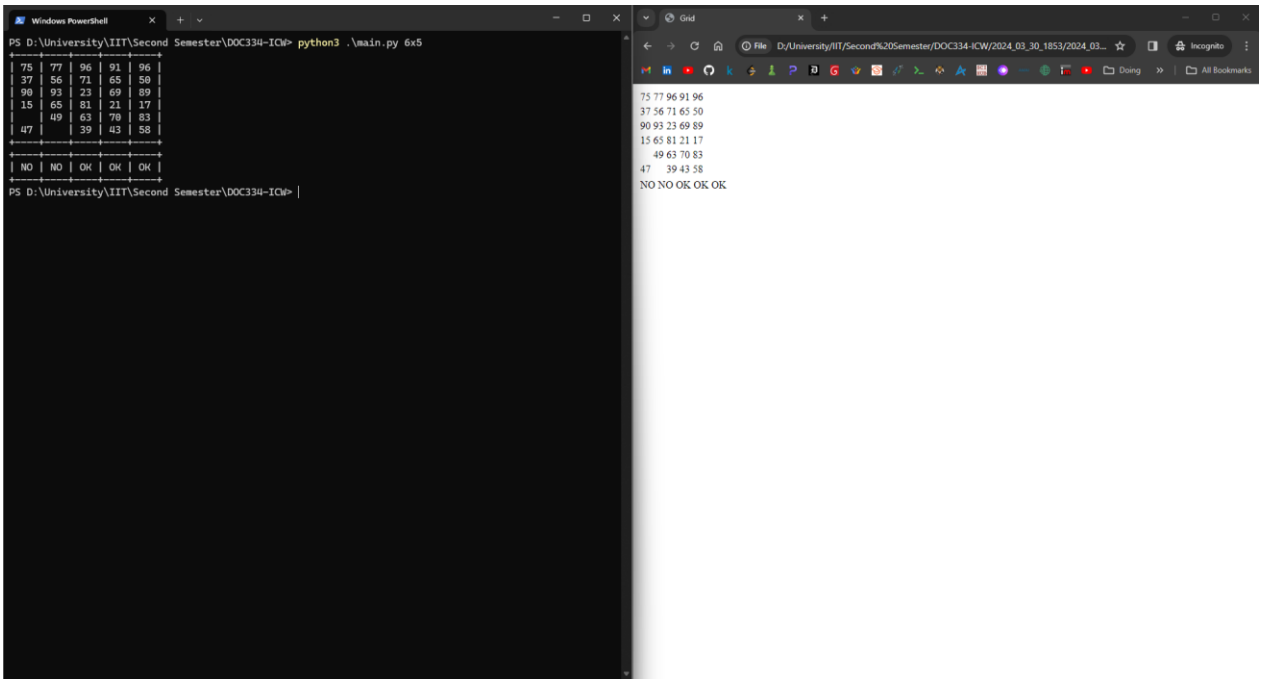


*Figure 33*

# 6 Future Improvements

The current percolation simulation program has implemented a basic level implementation, the program has been built with scalability, maintenance, and future implementation in mind. The following are a few possible future features that could be implemented to the current program:

- Enhanced Visualization: Create more advanced visualizations using libraries such as Matplotlib or Plotly. This would offer a better way for users to engage with the results.
- Parameterization: Allowing users to add more console arguments to adjust more parameters such as percentage (%) of empty cells, range of random numbers.
- Export to Other Formats: Adding the ability to export results to more formats such as CSV or Excel for further analysis.

# 7 Conclusion

In conclusion, the Python program developed achieved the objectives set out of simulating the percolation process. Grids are generated with random two digital numbers and random empty cells, the program mimics real world scenarios of liquid passing through filters. The program can produce HTML and Text Files as well. Overall, the program is a useful tool to study and understand percolation.