Assignment 2

Instructions

- Unless specified otherwise,
  - you can use anything in the Python Standard Library;
  - don't use any third-party library including NumPy in your code for this assignment;
  - you can assume all user inputs are always valid and require no validation.
- The blue texts in the Sample Runs section of each question refers to the user inputs. It does NOT mean that your program needs to underline them.
- Please follow closely the format of the sample output for each question. Your program should produce **exactly** the same output as the sample (same text, symbols, letter case, spacing, etc.). The output for each question should end with a single newline character, which has been provided by the `print()` function by default.
- Please follow exactly the specified name, parameter list and return value(s) for the functions to be written if a question has stated (for easier grading). You may define additional functions for specific tasks or further task decomposition in any question if you see fit.
- Name your script files as instructed in each question (case-sensitive). Using other names may affect our grading and may result in slight deduction.
- Your source files will be tested in script mode rather than interactive mode.
- You are highly encouraged to annotate your functions to provide type information for parameters and return values, and include suitable comments in your code to improve the readability.

Question 1 (15%)

Write a Python script `alphabattle.py` to implement a simple two-player game session.

The two players randomly put the 26 letters of the English alphabet in a string and ten integers (ranged from 0 up to 25) in a list. Each integer represents the index of a letter to eliminate from the opponent's string so that the two players will begin an "alphabet battle" with two reduced strings of 16 letters.

The letter removal phase is illustrated by the following example:

```
#    01234567890123456789012345
p = "MZNHUVIOEPTWFJCBXKALSDQGYR" # The 26 letters of Player P
q = "YFTUCSQOMGKPXNDWHIVJRABZEL" # The 26 letters of Player Q

pi = [1, 3, 2, 8, 10, 12, 9, 7, 4, 22]
# The indices to eliminate letters in string q:
# q[1] is "F", delete it!
# q[3] is "U", delete it!
# and so on ...

qi = [21, 24, 25, 3, 4, 1, 8, 9, 10, 17]
# The indices to eliminate letters in string p:
# p[21] is "D", delete it!
# p[24] is "Y", delete it!
# and so on ...

p = "MNVIOWFJCBXALSQG"
# Player P now has 16 letters
# The relative order is the same as before the elimination.

q = "YSQPNDWHIVJRAZEL"
# ... and so is for Player Q
```

Then run the battle: compare each pair of letters of the players' strings in terms of their ASCII values. During this process, the letter with the greater ASCII value gives to the possessing player a number of points equal to its value less the one of its opponent. If the letters are equal for both players, they don't gain any points. Let us continue the example to visualize this process as follows.

```
p = "MNVIOWFJCBXALSQG"
q = "YSQPNDWHIVJRAZEL"

# Each letter has an ASCII value, listed below from left to right:
ascii_P = [77, 78, 86, 73, 79, 87, 70, 74, 67, 66, 88, 65, 76, 83, 81, 71]
ascii_Q = [89, 83, 81, 80, 78, 68, 87, 72, 73, 86, 74, 82, 65, 90, 69, 76]

# Then each pair of ASCII values is compared among the players

# Round 1: 77 vs. 89
# Player Q wins and obtains 89 - 77 = 12 points

# Round 2: 78 vs. 83
# Player Q wins and obtains 83 - 78 = 5 points

# Round 3: 86 vs. 81
# Player P wins and obtains 86 - 81 = 5 points

# Round 4: 73 vs. 80
# Player Q wins and obtains 80 - 73 = 7 points

# Round 5: 79 vs. 78
# Player P wins and obtains 79 - 78 = 1 point

# and so on ...
```

In your script, write a function `alpha_battle(p, pi, q, qi)`, where `p` and `pi` are the string of the 26 distinct, uppercase letters and list of the 10 distinct integer indices in range [0, 25] of the player P respectively; `q` and `qi` are those of the player Q.

The function returns a two-entry dictionary denoting the scores of the players, for example:

```
# Result of the step-by-step example game:
{ "P": 64, "Q": 96 }
```

In your main program (under the “module name check” if-statement), prompt the user to enter an integer as seed for the random number generator, generate the random letter strings and index lists, call the above function, and print the result of the game.

Note: Your main client code will also be graded for this question. Depending on how you generate the random strings and lists, your random strings and lists could be different from the samples below. This is not a problem. For your reference, the sample output was obtained by using the `random.sample()` function to generate `p`, `q`, `pi`, `qi` in order.

Sample Runs

```
Enter seed: 1↵
Player P: ESCIDPOUMGVHAQRJNTZFBWKLYX [23, 10, 25, 22, 16, 13, 21, 6, 9, 17]
Player Q: AURZMGNWQHOXIDFVKPESJTYLBC [18, 15, 16, 12, 25, 1, 24, 7, 22, 13]
Score: {'P': 51, 'Q': 91}
Player Q wins!
```

```
Enter seed: 2↵
Player P: BCYLFJIGZVNKSMWTQOHEXAPRUD [14, 5, 12, 22, 25, 20, 16, 7, 15, 8]
Player Q: MNQFRWHTAUKCOIZXLJYDGESPBV [15, 16, 24, 11, 21, 14, 20, 22, 17, 19]
Score: {'P': 72, 'Q': 56}
Player P wins!
```

```
Enter seed: 208↵
Player P: IQGBSVZUOTRFXCMDEYLKHNPJAW [20, 11, 24, 17, 7, 25, 22, 12, 5, 13]
Player Q: TREYMHBSPLUOIQKVCDAFWJGNZX [15, 0, 12, 1, 2, 17, 21, 4, 7, 18]
Score: {'P': 81, 'Q': 81}
Draw game!
```

```
Enter seed: 506↵
Player P: YPIMSNXHQFURVCBAGWKELOJZTD [13, 15, 19, 9, 12, 23, 1, 14, 16, 10]
Player Q: DXROUZTPLVGIMNAKCHBFQWESYJ [21, 20, 19, 13, 9, 10, 3, 0, 2, 18]
Score: {'P': 60, 'Q': 60}
Draw game!
```

Question 2 – Word Analysis on the Holy Bible (25%)

The Holy Bible has 66 books and 1,189 chapters. For a typical English translation, these are made up of roughly 31,100 verses and 800,000 words. We are going to study which words are the top 10 in terms of frequency of occurrence in the Bible by writing a Python script to count words.

Write a Python script `bible.py` to download the text of the Holy Bible from a website and report the top 10 words having the highest frequencies of occurrence in the text.

The website is getbible.net which provides a web API for downloading bible passages. The API (v.2) can be called in three ways:

Download Type	URL	Example	What's Downloaded
The whole bible	https://getbible.net/v2/{ver}.json	https://getbible.net/v2/kjv.json	The whole King James Version (KJV) Bible
A book	https://getbible.net/v2/{ver}/{bk}.json	https://getbible.net/v2/kjv/1.json	The first book (Genesis) of KJV Bible
A book chapter	https://getbible.net/v2/{ver}/{bk}/{ch}.json	https://getbible.net/v2/kjv/1/3.json	Chapter 3 of Genesis of KJV Bible

where `{ver}`, `{bk}`, and `{ch}` are the bible version (a.k.a. translation), book id (1-66) and chapter id (its range depends on which book is in the query).

More information about this API can be found on this GitHub page, which has listed the available bible translations, books and book chapters near the bottom of the page.

The HTTP response is in JSON format and has different levels of attributes for the results of the three different queries. You need to study them carefully to see how to decode the JSON response into a flattened string containing the text of the whole bible, or a book or a book chapter.

In the script, define the following functions:

- `get_text(ver="akjv", bk=None, ch=None) -> str:`

  This function downloads the requested JSON via the API, decodes it into a dictionary, and returns a string which is created by concatenating all the "`text`" attribute values of all items of the "`verses`" attribute under each chapter of each book, with a space added between verses.

The default argument for `ver` (bible version) is "akjv" (American King James Version). If `bk` (book) and `ch` (chapter) are both None, the function downloads the whole bible (containing all the 66 books) of the specified version. If `bk` is specified, it downloads the specified book. If `ch` is also specified, it downloads the specified chapter of the specified book.

- `top_words(text: str, n: int = 10) -> dict[str, int]:`

  This function accepts a string (containing the bible text) and returns a dictionary containing (at most) n items denoting the top (most frequent) n words found in the string. The dictionary keys and values are distinct words (`str`) and their frequencies (`int`). The dictionary to return is sorted in **descending order** of values (i.e., by frequencies) instead of keys.

  We describe how the dictionary of top n words is generated as follows:

  1. **Split the text** on whitespace into a list of word tokens.
  2. For each word token in the list, **strip all leading and trailing punctuation** if any. For example, the token "Day," will become "Day" with the trailing comma stripped off. But the hyphens in the token "father-in-law" won't be removed. Note that some bible translations may contain non-ASCII Unicode punctuation such as "'" in the token "sons'". Ignore these cases and just keep them in the word tokens without stripping, so "sons" and "sons'" are treated as two different words. You can use the `string.punctuation` literal in the `string` module to facilitate this task.
  3. **Filter out stop words in the list**: Many words like 'a', 'an', 'the', 'of', 'for', 'that', etc. bear no meaning for our analysis. These are called *stop words*. Remove them before counting words. For this task, use our provided file `stopwords.txt`, which contains common stop words separated by whitespace (this stop word list is borrowed from MySQL's default stopwords for MyISAM search indexes). Assume that this file is in the current directory and its filename can just be hardcoded in your script. This filtering step should be done in a case-insensitive manner, i.e., the words "an", "An" and "AN" should all be removed.
  4. **Remove all single-letter words** such as "I" and "a" too.
  5. Based this preprocessed word list, you can build the required dictionary with distinct words as keys and their occurrence counts as values.
  6. Sort the dictionary by value (occurrence count) in descending order.
  7. Return a sliced version of the dictionary with the first n elements only.

Note: You don't need handle stemming and lemmatization in this question as one would do in natural language processing. So, "come", "came", "coming" will become different keys with their own word counts as values in the output dictionary. The same applies to letter case and plural forms, so "God" and "god", "son" and "sons", "man" and "men" are taken as different words.

Restrictions

As stated in the beginning, don't use any third-party library such as NLTK to attempt this question. But the `requests` module is an exception – you are allowed to use it in this question.

Sample Runs

A sample main client code is provided as follows. You may use it for testing and are surely encouraged to modify it to include some more test cases. Put your testing code under the "module name check" if statement, which won't be graded. We will import your module into our main program to test and grade the required functions stated above.

```
if __name__ == "__main__":
    versions = ["akjv", "kjv", "web"]
    torah = {"Gen": 1, "Ex": 2, "Lev": 3, "Num": 4, "Deut": 5}
    gospel = {"Matt": 40, "Mark": 41, "Luke": 42, "John": 43}

    scripture = get_text("web", torah["Gen"], 1)  # chapter
    words = top_words(scripture)
    print("Gen1:\n", words)

    for g in gospel:  # book
        scripture = get_text(bk=gospel[g])
        words = top_words(scripture)
        print(g + ":\n", words)

    for v in versions:  # bible
        scripture = get_text(v)
        words = top_words(scripture)
        print(v + ":\n", words)
```

The expected output of this code snippet is given as follows:

```
Gen1:
 {'God': 30, 'earth': 14, 'waters': 10, 'light': 9, 'day': 9, 'expanse': 9, 'kind': 9,
'“Let': 8, 'good': 7, 'sky': 7}
Matt:
 {'Jesus': 169, 'man': 124, 'heaven': 75, 'disciples': 71, 'things': 57, 'men': 56,
'kingdom': 56, 'Lord': 55, 'God': 55, 'Son': 49}
Mark:
 {'Jesus': 97, 'man': 87, 'God': 51, 'things': 47, 'disciples': 46, 'house': 30,
'answered': 30, 'cast': 28, 'John': 26, 'eat': 26}
Luke:
 {'man': 127, 'God': 124, 'son': 104, 'things': 99, 'Jesus': 98, 'Lord': 87, 'day': 63,
'pass': 58, 'people': 58, 'house': 58}
John:
 {'Jesus': 255, 'man': 125, 'Father': 114, 'God': 82, 'world': 80, 'answered': 77,
'things': 70, 'Jews': 66, 'disciples': 66, 'life': 44}
akjv:
 {'LORD': 6543, 'God': 4086, 'man': 2600, 'Israel': 2565, 'king': 2181, 'people': 2139,
'son': 2085, 'house': 2024, 'children': 1798, 'day': 1733}
kjv:
 {'Lord': 7595, 'thou': 4890, 'thy': 4453, 'God': 4384, 'thee': 3826, 'ye': 3703,
'man': 2602, 'Israel': 2565, 'hath': 2238, 'king': 2181}
web:
 {'Yahweh': 5745, 'God': 3659, 'Israel': 2474, 'man': 2139, 'king': 2086, 'son': 2026,
'house': 1880, 'people': 1850, 'children': 1776, 'land': 1727}
```

Note: The above shows the output dictionary on 2 lines just because of limited width of this document; in the actual output, there is no line break in the middle when printing the dictionary.

The above main client code may take a few ten seconds to finish (depending on the server load of the API website, the speed of the network and your computer, and somehow your algorithm as well). While performance is not an important grading criterion in this question, if the execution is much (e.g., several times) longer, this may imply your code is likely using a too-slow algorithm for processing the text and data structures involved. In this case, you should check for the reasons behind and revise your code accordingly.

Question 3 – The Reversi Game (60%)

Write a Python script `reversi.py` to simulate the Reversi board game.

Reversi is a two-player game which is played on a square uncheckered board with some game pieces. A piece is a *disc* with two sides in black and white colors respectively. At the beginning of a game, there are four pieces placed on the board as shown in Figure 1. We use a dot '.' to denote an empty cell in the board, the character 'X' to denote a black piece, and the character 'O' to denote a white piece. To denote a position in a game board, we use a coordinate system like a spreadsheet program, e.g. "A2" refers to the cell in the first column and the second row. Suppose that a nested list `board` (let's call it a 2D array) is used to represent the game board, cell address "A2" will be mapped to `board[1][0]`.

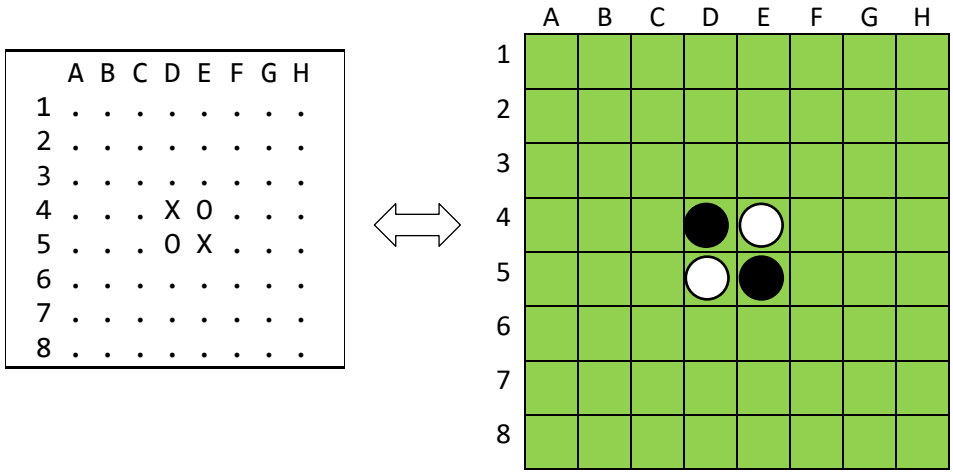

Figure 1: Initial configuration of the game board

The two players, namely "Black" and "White" make moves in turns by putting a disc of their color on an empty cell of the board. Black plays the first move. A move cannot be made arbitrarily to any of the empty cells. As a *valid move*, the newly placed disc must be in a position such that it forms *at least one straight (horizontal, vertical, or diagonal) occupied line between the new piece and another piece of the same color, with one or more contiguous pieces of the opposite color between them*. After the move, those contiguous pieces will all be *flipped* to become the color of the new piece. Figure 2 shows an example move made by White. Note that a move can simultaneously form more than one straight line. In such a case, the sandwiched pieces on *all* the lines formed will be flipped. Figure 3 shows such an example.

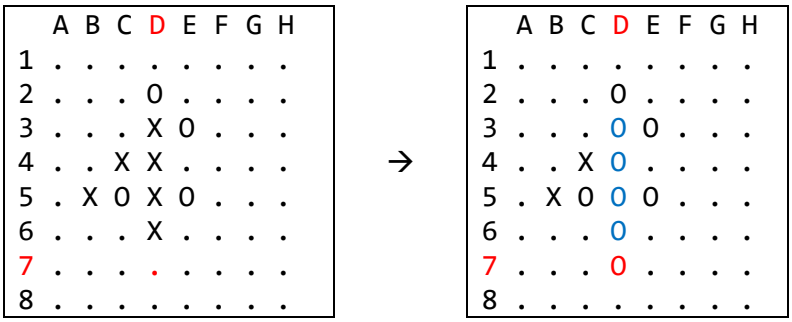

Figure 2: An example move by White ('O') at position D7, forming a vertical occupied line from D7 to D2. Four black ('X') pieces are "clipped" by two white pieces at both ends and are flipped.

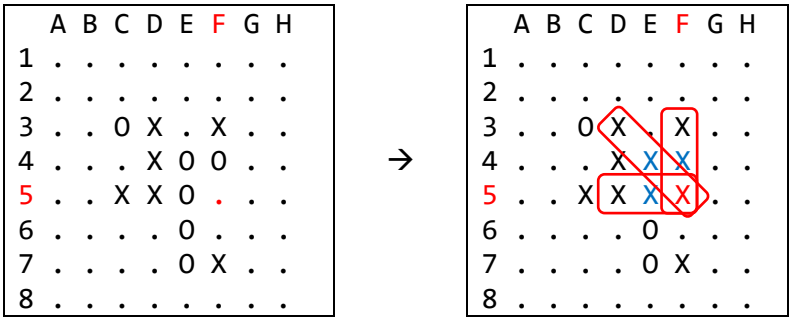

Figure 3: An example move by Black at F5, forming three occupied lines
(i) F5 to F3, (ii) F5 to D5, and (iii) F5 to D3. All sandwiched pieces of white are flipped to black.

A game is over when either the board is full or both players have no valid moves. The player having more pieces of his/her color than the opponent wins the game. If both players have the same number of pieces, the game draws.

To add more challenges to the game, this assignment will extend the original Reversi game with an additional feature: some *"blocks"* can be added to the game board, preoccupying some cells so that both players cannot place a disc onto such positions. Blocks are added before the game begins. For example, Figure 4 shows a board with cells A1, A2, B1, B2, G5, G6, G7 and F7 initialized as blocks (denoted by the '#' symbol). Let's call them "block cells". Then players cannot make a move to such positions. Apart from fewer choices for making a move and greater likelihood that the players run out of valid moves before the board becomes full, the overall rules of the game stay the same as in the case without block cells.

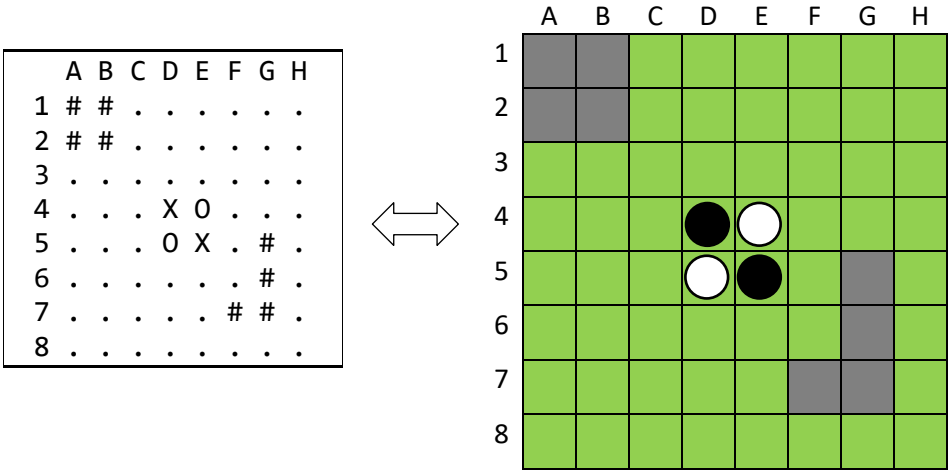

Figure 4: Initial configuration of the game board with blocks (grey areas) added

Game board

You must use a nested list of `str` elements (a 2D array) to represent the game board.

Note that defining the board array as a global variable is forbidden and will invite mark deduction if you do so. Define it as a local variable in the main function of the program.

You should also avoid defining it in a hard coded manner; make sure the board size is scalable, it is not always fixed at 8 x 8.

Basic Game Flow

- The program needs to set up the game board first. Before a game begins, the program will prompt the user to enter the number of block cells to add to the board. It can be zero or up to half of the board's area at most. For example, if N (board size) = 8, the maximum number of block cells allowed would be (8 * 8) / 2 = 32. This ensures that the board still has enough empty cells for making moves. It is illegal to set any of the initial four pieces central in the board as a block. Entering a cell address that refers to a block already defined or a cell out of the board's size range is illegal too. If the user enters an exceedingly value for N or any illegal cell addresses, the program repeats prompting until valid inputs are received.
- Then the game starts with round 1 on the board initialized with the first four discs as shown in Figure 1 and optionally with block cells. Black moves first.
- Black and White make moves alternately. For each move, the program should prompt the player to enter the cell address of the intended move, e.g., "D3", and it will convert the input string into two integer values (x, y) that represent the column and row indexes, respectively, to access the corresponding element of the array board[y][x].
- You should check whether a player's input refers to a valid move. If the input position is not valid (either because the input coordinate is out of bounds of the board or because the input position cannot flip any opponent's pieces), the player should be prompted again for a valid move.
    - In fact, it can happen that the board with empty cells has really run out of positions for a player to make a valid move. Thus, your program needs to check whether there still exists at least one empty cell that allows the current player to make a valid move before prompting him/her for entering a cell address. This involves scanning the whole game board for any cells for a valid move per round before prompting.
    - In case of no possible valid moves, the player's round will be "passed" to the opponent.
- If a valid move is made, all relevant pieces should be flipped to the opposite color.
- The game should *end when either (1) the board is full, or (2) two consecutive passes are made (i.e., Black passes to White but White immediately passes back to Black since both of them have no valid moves to make)*. If Black wins, the message "Player X wins!" should be printed; if White wins, the message "Player O wins!" gets printed; and if the game is a draw, "Draw game!" should be printed.

Decomposition into Functions

Your program should be *reasonably decomposed into different functions* for various tasks such as printing the game board, initializing the board, adding blocks, checking if a move is valid, and updating the board to realize a move, etc. The game board array and other data like the input coordinate values can be passed between the functions.

To ease our grading, your program must implement the *required functions below, which will be graded individually via unit testing.* That is, we will import your reversi module into our main module containing our designed test cases testing the required functions one by one. So your code for each function shall implement the description of that function only. You shall *not* write any code in a function that is beyond that function's description.

(1)
```
valid_move(board: list[list[str]], p: str, y: int, x: int, flip: bool = False)
    -> bool | None
```

Return true if the move to be made by player p at cell (y, x) is valid, or false otherwise, and None if flip is true.

This function has two usages: (1) check if a move is valid; (2) carry out the move on the board. It has an optional Boolean parameter flip (default to false). If the flip argument is passed as true, it will carry out the move on the board, i.e. flipping the opponent's discs in all possible directions, instead of checking.

For better code readability, you may optionally create a *wrapper* function named flip_discs() or make_move(), etc., which calls this function with flip passed as true. Then your main function can call valid_move() to check move validity and call the wrapper function to make a move on the board respectively.

(2) **`has_valid_moves(board: list[list[str]], p: str) -> bool`**

Return true if player p still has at least one valid move on the board, or false otherwise.

(3) **`has_empty_cells(board: list[list[str]]) -> bool`**

Return true if the game board still has at least one empty cell, or false otherwise.

(4) **`main() -> None`**

Define the 2D array for the game board and implement the main game flow by prompting the user for inputs and calling the functions that you define. Your program must have an if statement below to call this function.

```
if __name__ == "__main__":
    main()
```

Assumptions

- The board is always a square. And the board size is always even. So, we won't have 5x5, 7x7, 9x9, … boards but 4x4, 6x6, 8x8, … boards only.
- The minimum and maximum board sizes are 4 and 26 respectively. So, the largest column is up to Z only. You need not spend too much time on testing big game boards (up to 12 is enough).
- The row index markers on the left of the printed board are always right adjusted at a fixed width of 2 characters, i.e., for single-digit row indexes, there is a single space ahead of them. For example, Figure 5 shows a board with N = 10. Cell addresses being entered will still be like "A1", "B8", "C10", etc. No zero padding is used, so the expected inputs are not like "A01", "B08", etc.
- We assume that cell address inputs always follow the format of one letter (A-Z or a-z) plus one integer (1-26). Lowercase inputs like "a1", "h10", etc. will be accepted as normal. You need NOT handle weird inputs like "AA1", "A01", "abc", "A3.14", "#a2", … for cell addresses and inputs like "-1", "6.28", "#$@", … for the number of blocks. The behavior upon receiving these is unspecified, and we won't test your program against these inputs.

```
   A B C D E F G H I J
 1 . . . . . . . . . .
 2 . . . . . . . . . .
 3 . . . . . . . . . .
 4 . . . . . . . . . .
 5 . . . . X O . . . .
 6 . . . . O X . . . .
 7 . . . . . . . . . .
 8 . . . . . . . . . .
 9 . . . . . . . . . .
10 . . . . . . . . . .
```

Figure 5: Initial configuration of a game board with 10 rows and 10 columns

Restrictions

- Don't use global variables as far as possible; pass data between functions using parameters and/or return values. If you define any *global variables*, they must be read-only instead of read-write.
- *Avoid defining your own classes* in this assignment. We will have assignments on OOP to come.

Sample Runs

In the following sample runs, the blue text is user input and the other text is the program printout. You can try the provided sample program for other input.

Sample Run #1: (for N = 6)

```
Enter board size: 6↵
Enter number of blocks: 19↵
Too many blocks!
Enter number of blocks: 7↵
Enter position for block 1: A1↵
Enter position for block 2: A2↵
Enter position for block 3: B1↵
Enter position for block 4: B2↵
Enter position for block 5: C3↵
Invalid position!
Enter position for block 5: D4↵
Invalid position!
Enter position for block 5: X10↵
Invalid position!
Enter position for block 5: A1↵
Invalid position!
Enter position for block 5: B5↵
Enter position for block 6: E5↵
Enter position for block 7: E2↵
Round 1:
   A B C D E F
 1 # # . . . .
 2 # # . . # .
 3 . . X O . .
 4 . . O X . .
 5 . # . . # .
 6 . . . . . .
Player X's turn: E3↵
Round 2:
   A B C D E F
 1 # # . . . .
 2 # # . . # .
 3 . . X X X .
 4 . . O X . .
 5 . # . . # .
 6 . . . . . .
Player O's turn: C2↵
```

…
(some rounds skipped due to space constraint)
…

```
Round 17:
   A B C D E F
 1 # # X O . .
 2 # # O O # X
 3 O O X O O X
 4 O O O O O X
 5 . # O O # .
 6 . . O . . .
Player X's turn: E1↵
Round 18:
   A B C D E F
 1 # # X X X .
 2 # # O X # X
 3 O O X O O X
 4 O O O O O X
 5 . # O O # .
 6 . . O . . .
Player O has no valid moves! Pass!
Round 19:
   A B C D E F
 1 # # X X X .
 2 # # O X # X
 3 O O X O O X
 4 O O O O O X
 5 . # O O # .
 6 . . O . . .
Player X's turn: A5↵
Round 20:
   A B C D E F
 1 # # X X X .
 2 # # O X # X
 3 O O X O O X
 4 O X O O O X
 5 X # O O # .
 6 . . O . . .
```

…
(some rounds skipped due to space constraint)
…

```
Round 24:
   A B C D E F
 1 # # X X X .
 2 # # O X # X
 3 O O X X O X
 4 O X O X X X
 5 O # X O # X
 6 O . O O O .
Player O's turn: B6↵
Round 25:
   A B C D E F
 1 # # X X X .
 2 # # O X # X
 3 O O X X O X
 4 O X O O X X
 5 O # O O # X
 6 O O O O O .
Player X has no valid moves! Pass!
Round 26:
   A B C D E F
 1 # # X X X .
 2 # # O X # X
 3 O O X X O X
 4 O X O O X X
 5 O # O O # X
 6 O O O O O .
Player O has no valid moves! Pass!
Game over:
   A B C D E F
 1 # # X X X .
 2 # # O X # X
 3 O O X X O X
 4 O X O O X X
 5 O # O O # X
 6 O O O O O .
Player O wins!
```

Sample Run #2: (for N = 4)

(This example is an extreme case with so many block cells that no valid moves can ever be made since the start of the game.)

```
Enter board size: 4↵
Enter number of blocks: 9↵
Too many blocks!
Enter number of blocks: 8↵
Enter position for block 1: B1↵
Enter position for block 2: C1↵
Enter position for block 3: A2↵
Enter position for block 4: D2↵
Enter position for block 5: A3↵
```

```
Enter position for block 6: D3↵
Enter position for block 7: B4↵
Enter position for block 8: C4↵
Round 1:
   A B C D
 1 . # # .
 2 # X O #
 3 # O X #
 4 . # # .
Player X has no valid moves! Pass!
Round 2:
   A B C D
 1 . # # .
 2 # X O #
 3 # O X #
 4 . # # .
Player O has no valid moves! Pass!
Game over:
   A B C D
 1 . # # .
 2 # X O #
 3 # O X #
 4 . # # .
Draw game!
```

- There are many combinations of possible inputs. Please check your program correctness against the results produced by our sample program executable posted on Blackboard.

Sample Executable

For more sample runs, you can execute our provided modules in binary format:
- `reversi.so` (for macOS, arm64 or Intel x64)
- `reversi.pyd` (for Windows, arm64 or Intel x64)

Download and put the executable binary file that fits your OS in your current directory. Start a Python shell and type the following statement to start the execution of the sample program.

```
>>> import reversi
>>> reversi.main()
```

Note that the executable was built in a Python 3.10 environment. You need a Python runtime of that version or later to execute the binary. If you are using Anaconda or Miniconda, which is currently using Python 3.9 as the base environment, please create a virtual environment installing Python 3.10 to run the executable.