Connect Four Game

Learning Goals

In this project, you will learn to:

- practice using nested loops and conditionals to iterate over a list of strings
- break down a larger program into several smaller functions to make it easy to implement
- create printouts that match the required format exactly

Note: Please read this document carefully and thoroughly. There are several details in this assignment that require you to write your program exactly as directed.

Background

The goal of this project is to implement the game Connect Four. Your program will be the host allowing two human players to play the game.

Connect Four

Traditionally, the connect four game is a two-player game in which each player first chooses a color and then takes turns dropping colored discs from the top into a seven-column, six-row vertically suspended grid. The pieces fall straight down, occupying the lowest available spot in the chosen column. The goal is to be the first to form a line of four (either horizontal, or vertical, or diagonal) of one's own discs. Please see the Wikipedia entry for more details.

Game Board and Moves

The following represents a 6-row by 7-column grid. The rows are indexed (from top to bottom) 0 to 5 and the columns are indexed (from left to right) 0 to 6. For now we use a dot `.` to represent an empty spot on the board. So an empty board looks like this:

```
   0 1 2 3 4 5 6
0  . . . . . . .
1  . . . . . . .
2  . . . . . . .
3  . . . . . . .
4  . . . . . . .
5  . . . . . . .
```

For simplicity (as we are not drawing graphics in this project), we will use characters `X` and `O` to represent the two colors of discs. Below is an example state of the game. We assume player `X` always makes a move first.

```
   0 1 2 3 4 5 6
0  . . . . . . .
1  . . . . . . .
2  . . O . . . .
3  . . O X . . .
4  . O X O X . .
5  . O X X O X .
```

The sequence of moves can be represented as a list such as `[3,4,2,1,5,1,2,2,4,3,3,2]` where each number is the column index that a player (beginning with player `X`) has chosen to drop their disc. In this example, player `X` chose column `3` first, then player `O` chose column `4`, then player `X` chose column `2` and so on.

Game Play

Continuing from the above game state, if player `X` chooses column `1` in the next move, the resulting state is shown in the first board below. We use a **`blue bolded`** font to indicate where the new disc appears. Let's say the next three moves are: player `O` chooses column `0`, player `X` chooses column `6`, and finally player `O` chooses column `3`. At this point, player `O` has won the game, because we've got a *connect four* in a diagonal line connecting positions `(5,0), (4,1), (3,2), (2, 3)` (indicated by red fonts in the last board below). The game is now over.

```
   0 1 2 3 4 5 6  |    0 1 2 3 4 5 6  |    0 1 2 3 4 5 6  |    0 1 2 3 4 5 6
0  . . . . . . .  | 0  . . . . . . .  | 0  . . . . . . .  | 0  . . . . . . .
1  . . . . . . .  | 1  . . . . . . .  | 1  . . . . . . .  | 1  . . . . . . .
2  . . O . . . .  | 2  . . O . . . .  | 2  . . O . . . .  | 2  . . O O . . .
3  . X O X . . .  | 3  . X O X . . .  | 3  . X O X . . .  | 3  . X O X . . .
4  . O X O X . .  | 4  . O X O X . .  | 4  . O X O X . .  | 4  . O X O X . .
5  . O X X O X .  | 5  O O X X O X .  | 5  O O X X O X X  | 5  O O X X O X X
```

Assignment

You will write several functions to implement the connect four game. To generalize the game, we allow the game board size to be flexible, where the number of rows and columns can be anywhere between `[4,10]`, so the board can be as small as `4x4`, and as large as `10x10`.

For this project, you **must work alone**. You are welcome to talk to other students, but you **must write code on your own and submit on your own**. **NO group submission** is allowed.

Tip: you will find it extremely helpful to draw the game board on a piece of paper, make a few moves manually to help you understand how the game works and plan out your program.

0. (Common Paragraph) Create the file `connect4.py`

Create a new file named `connect4.py` and at the top of this file include your **name, email, and Spire ID**. Do NOT include other students' information as you must submit individually. Example:

```
# Author   : Bart Bartly
# Email    : bbartly@umass.edu
# Spire ID : 29384757
```

1. Implement function `make_empty_board` (10 points)

Write a function called `make_empty_board`. It should take two parameters: the number of rows (`nrows`) and number of columns (`ncols`), and **return a list of strings** representing the board of the requested number of rows and columns. You should use a dot `.` to represent an empty spot. Specifically, the returned list should have `nrows` strings, each containing `ncols` dots. You can assume the number of rows and the number of columns are both between 4 and 10. Do **NOT** ask the user for any input, do **NOT** print anything in this function. An example is given below:

```
print(make_empty_board(6, 4)) # 6 rows and 4 cols
>>>  ["....", "....", "....", "....", "....", "...."]
```

Hint: this function is easy to implement and does not even need a loop. Recall that for any string or list objects, you can multiply it by an integer to replicate it that many times.

2. Implement function `print_board` (15 points)

Write a function called `print_board`. It should take one parameter: a list of strings representing the game board, and it should **print** the board (note: print, **NOT** return) in the specified format below. In the input list of strings, a character `X` represents an `X` disc, a character `O` represents an `O` disc, and a dot `.` represents an empty spot. You can assume there are no other characters.

- Each spot on the board should be printed by 3 characters: space-disc-space. Specifically, an `X` disc is printed as `" X "`, an `O` disc as `" O "`, and an empty spot (i.e. a dot in the input strings) as `"   "` (i.e. 3 space characters).
- Every two spots in the same row should be separated by a character `"|"`. There should be NO trailing `"|"` after the last spot in a row.
- Every two rows are separated by sequences of `"---+"` as shown in the examples below.

Note: the printouts of your function must match the format above exactly, and be identical to the examples below. Any extra or missing character will cause the autograder tests to fail.

You can assume the list of strings passed to this function is a valid game board, in that every string in the list has the same length, and there are no other characters beyond `'X'`, `'O'`, and `'.'`. As you can see, the length of the list is the number of rows, and the length of each string is the number of columns. Below is the expected printout if we call `print_board` with an empty game board of dimension 5 x 6:

```
print_board(make_empty_board(5, 6))
>>>
   |   |   |   |   |
---+---+---+---+---+---
   |   |   |   |   |
---+---+---+---+---+---
   |   |   |   |   |
---+---+---+---+---+---
   |   |   |   |   |
---+---+---+---+---+---
   |   |   |   |   |
```

As another example, below is an example printout of a non-empty board of dimension 6 x 7:

```
print_board([".......", ".......", "..O....", "..OX...", ".OXOX..",".OXXOXX"])
>>>
   |   |   |   |   |   |
---+---+---+---+---+---+---
   |   |   |   |   |   |
---+---+---+---+---+---+---
   |   | O |   |   |   |
---+---+---+---+---+---+---
   |   | O | X |   |   |
---+---+---+---+---+---+---
   | O | X | O | X |   |
---+---+---+---+---+---+---
   | O | X | X | O | X | X
```

Hint: to avoid printing a trailing character, you can use an `if` statement in your loop to avoid printing the character at the last iteration. Think about how to check for the last loop iteration.

3. Implement function `verify_board` (20 points)

Write a function called `verify_board` that takes one parameter: a list of strings representing the game board. It should return `True` if the game state is valid, and `False` otherwise.

A valid game state should adhere to the following rules:

1. Because players take turns to play, the number of `X` discs and `O` discs in any game board should not differ by 2 or more. For example, given the following board, `verify_board` should return `False` because there are 6 `O`s but only 4 `X`s, indicating an invalid state.

```
board = [".......", ".......", "..O....", "..OX...", ".O.OX..",".O.XOX."]
print(verify_board(board))
>>> False
```

2. Since the discs fall from the top to the lowest empty spot, any disc, except the last row, must have another disc in the same column underneath it. Otherwise, if a disc has no other disc underneath it and is not in the last row, it would defy gravity resulting in a hanging disc. For example, given the following board, `verify_board` should return `False` because the first `X` is in an invalid state (with an empty spot underneath it).

```
board = ["X......", ".......", "..O....", "..OX...", ".O.OX..",".O.XOXX"]
print(verify_board(board))
>>> False
```

4. Implement function `verify_move` (15 points)

Write a function called `verify_move`. It takes two parameters: a list of strings representing the game board and an integer index representing the column that the current player has chosen to drop disc into. It should return `True` if the specified column of the board is **NOT** full (i.e. row 0 at that column has no disc), and `False` otherwise. If a column is full, obviously you can not drop more discs into that column. In addition, this function should return `False` if the specified column index is out of bound (i.e. either `<0` or `>=ncols` in the game board)

Here are some example return values for the parameters given below:

```
board = ["..O....", "..X....", "..O....", "..OX...", ".OXOX..",".OXXOX."]
print(verify_move(board, 2))
>>> False
print(verify_move(board, 1)
>>> True
print(verify_move(board, 7)
>>> False
```

5. Implement function `update_board` (20 points)

Write a function called `update_board`. It takes **three** parameters: a list of strings representing the game board, an integer column index, and a string representing the player disc (`'X'` or `'O'`). It should **modify the input list** to reflect the board state after the player's disc falls into the specified column. Specifically, it should find the **lowest row** in the specified column that has an empty spot, then replace that dot by the player disc. At the end of the function, **return** the list. In other words, this function should modify the list that's passed in, and return it in the end.

For example, given the following board, column index `0`, and string `'X'` as arguments, update_board should modify the board and return it, resulting in its new state below. Blue font is used to mark where the disc is placed.

```
board = [".......", ".......", "..O....", "..OX...", ".OXOX..",".OXXOX."]
print(update_board(board, 0, 'X'))
>>> [".......", ".......", "..O....", "..OX...", ".OXOX..","XOXXOX."]
```

As another example, given the following board, column index 2, and the string 'O' as arguments, update_board should modify and return the board as shown below. Again, the player disc is placed at the lowest row in the specified column that hasn't been occupied.

```
board = [".......", ".......", "..O....", "..OX...", ".OXOX..",".OXXOX."]
print(update_board(board, 2, 'O'))
>>> [".......", "..O....", "..O....", "..OX...", ".OXOX..",".OXXOX."]
```

6. Implement function has_won (20 points)

Write a function called has_won. It should take two parameters: a list of strings representing the game board, and an integer representing a column index (of the most recent move). It should return True if the topmost disc (i.e. the one with the smallest row index) in the specified column leads to a winning state, and False otherwise. Because we can call has_won after each player makes a move, if there is a winning state, it must be a result of the most recent move. So it suffices to check whether the topmost disc in the specified column leads to a winning state.

For example, when provided with the following game board, and a column index 3 as arguments, has_won should return True. This is because the topmost disc in column 3 (in this case, disc O at row 2 column 3) forms a winning state by connecting all the discs at positions (2, 3), (3, 2), (4, 1), and (5, 0).

```
board = [".......", ".......", "..OO...", ".XOX...", ".OXOX..","OOXXOXX"]
print(has_won(board, 3))
>>> True
```

Note that a winning state is any 4 connected discs of the same character at the same column, or row, or diagonal (there can be two diagonal directions).

Hint: this function is the most difficult in this project because you need to consider multiple situations. To begin, find out where the **topmost** disc is in the **specified column**, and what character it is. We call this the **current disc** and denote its row as row_top and character as disc. To check for the winning state in the current column is the easiest, as there cannot be any disc above row_top, so it suffices to check if the three consecutive discs below row_top are exactly the same character as disc. If so, this column is in a winning state, and you can return True.

Next, to check for the winning state in the row is slightly more complex, as there can be discs both to the left and right of the current disc. To do so, think about all the characters in row_top. From the current column, go towards the left and count how many consecutive characters are exactly the same as `disc`, and stop counting if you either encounter any character that's not `disc`, or you have reached column 0. Similarly, from the current column, go towards the right and count how many consecutive characters are exactly the same as `disc`, and stop counting if you either encounter any character that's not `disc`, or you have reached the last column. If the total count (towards the left, towards the right, plus the current disc) is 4 or more, this row is in a winning state and you can return `True`.

Lastly, you need to check for winning states in the diagonals. This is similar to checking row winning states, except there are two possible diagonal directions: lower-left to upper-right (LL-UR), and upper-left to lower-right (UL-LR). Take the LL-UR direction as an example: from the current row (`row_top`) and column, go towards lower-left (i.e. each iteration row increments by 1 and column decrements by 1), and count how many consecutive characters are exactly the same as `disc`, and stop counting if you either encounter any character that's not `disc`, or either row or column index is out of bound. Do the same towards the upper-right direction. If the total count (including the current disc) is at least 4, this (LL-UR diagonal) is in a winning state. Finally do a similar check for UL-LR diagonal.

If none of the above checking leads to a winning state, return `False` in the end.

7. (Optional) Putting Everything Together and Play the Game

This part is optional (i.e. not graded). Now you have written all the functions as building blocks of the game, you can put everything together to play the game. Here are the steps:

1. Call `make_empty_board` to create a list of strings to represent an empty board.
2. Create a loop that iterates from `0` to `nrows*ncols` where `nrows` and `ncols` represent the number of rows and columns respectively. In the loop:
   a. Alternate between `X` and `O` for each iteration to represent the players taking turns.
   b. Prompt the user for their desired column index, then call `verify_move` to verify that the move is valid.
   c. If the move is invalid, print a prompt to the user and repeat step 2b until the player chooses a valid move.
   d. Call `update_board` to update the board with the validated move.
   e. Use `assert` with the `verify_board` function to ensure that the game board's status after 2d is valid.
   f. Display the current game board status using the `print_board` function.
   g. Call `has_won` function to check whether the move the player just made results in a winning position. If so, print congratulations and exit the loop.
3. At the end of the game, print a message indicating the winner, if there is one. Note that it's possible the game is a tie (i.e. no winner when all spots have been filled with discs).

These steps will allow you to create and play a basic Connect Four game using the functions you've implemented.

(Common Paragraph) Submit to Gradescope, Correct mistakes

Submit your program to gradescope. Wait for the autograder and review failed tests if any. Make the appropriate changes in your code and resubmit as many times as you need before the deadline. While working on each question, you can submit a partially finished program and receive feedback. Your highest scoring submission will be recorded as your grade. Note that each failed test includes text at the bottom giving hints as to why that test failed.