Problem 2: The Board Class

Connect Four is a variation of tic-tac-toe played on a 7x6 rectangular board.

The game is played by two players, alternating turns, with each trying to place four checkers in a row vertically, horizontally, or diagonally. One constraint in the game is that because the board stands vertically, the checkers cannot be placed into an arbitrary position. A checker may only be placed at the top of one of the currently existing columns (or it may start a new column).

In this problem, you will need to create a class named `Board` that implements some of the features of the Connect Four game. The `Board` class will have three data members: there will be a two-dimensional list (a list of lists) containing characters to represent the game area, and a pair of variables holding the number of rows and columns on the board. Six rows and seven columns is standard, but your `Board` datatype will be able to handle boards of any size.

Even as we allow arbitrary board sizes, however, we will preserve the four-in-a-row requirement for winning the game. Admittedly, this makes it hard to win the game on a 3x3 board.

The `Board` Class

Your `Board` class should have at least three data members:

- A variable `data` storing the two-dimensional array (list of lists), which is the game board
- A variable `height` storing the number of rows on the game board
- A variable `width` storing the number of columns on the game board

Note that the two-dimensional list is a two-dimensional list of *characters*, which are just strings of length 1. You should represent an empty slot by `' '`, which is the space character—not the empty string. You should represent player X's checkers with an `'X'` (the capital x character) and you should represent player O's checkers with an `'O'` (the capital o character).

Warning!

A **very** difficult bug to find occurs if you use the **zero** `'0'` character instead of the `'O'` character (capital-O) to represent one of the players. The problem occurs when you start comparing values in the board with the wrong one! Be sure to stay consistent with the capital-O character.

Starter Code

The starter code for this problem includes the `__init__` and `__repr__` functions that we wrote in class. However, **you will need to modify the `__repr__` function to include column numbers.** Get started by making a copy of this trinket:

`__init__`, the constructor

Signature: `__init__(self, width, height)`

This is a constructor for `Board` objects that takes two arguments. (Remember that `self` refers to the object being constructed and that it is not explicitly passed into the constructor.) This constructor takes in a number of columns and rows (7 and 6 are the connect-four standard, but our datatype will handle arbitrarily-sized boards). The constructor will set the values of the data members of the object. In addition, it will initialize the two-dimensional list representing the board. You will want to construct a list of lists ***very much in the spirit of the Game of Life problem***.

Note that you don't want `self.data` to contain all of the characters that go into printing the board—only the ones that affect the play of the game. The extra characters will be generated in the `__repr__` method.

`__repr__`, for printing or any string `repr`esentation

Signature: `__repr__(self)`:

This method returns a string representing the `Board` object that calls it. Basically, each "checker" takes up one space, and all columns are separated by vertical bars (|, typed by pressing shift and the backslash key between enter and delete on the keyboard). The columns are labeled at the bottom. Here is an example from the completed function for a 6-row and 7-column (6x7) board. You will need to replace the comment on line 32 of the provided starter code with code that produces the line numbers.

```
| | | | | | |
|
| | | | | | |
|
| | | | | | |
|
| | | | | | |
|
| | | | | | |
|
| | | | | | |
|
---------------
 0 1 2 3 4 5 6
```

In order to keep things in line, the column-numbering should be done "mod 10," as this larger 5-row, 15-column (5x15) example shows:

```
| | | | | | | | | | | | | | |
|
| | | | | | | | | | | | | | |
|
| | | | | | | | | | | | | | |
|
| | | | | | | | | | | | | | |
|
| | | | | | | | | | | | | | |
|
-------------------------------
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4
```

One string, multiple lines?

The string `'\n'` represents the *newline* character in Python. Thus, you can get multiple lines into a single string by adding `'\n'`. For example,

```
>>> s = 'This is the top line.'
>>> s += '\n'
>>> s += 'This is a second
line!\n'
>>> print s
This is the top line.
This is a second line!

>>>
```

Notice that the `'\n'` can be included *within* a string, as well. In this case, it added an extra blank line at the end of `s`
.

Next, implement the following methods in your Board class. Be sure to test each one after you write it—it's much easier to make sure each works one-at-a-time than trying to debug a huge collection of methods all at once. Be sure to try the hints on how to test each one after writing it!

`addMove`, for dropping a checker into the board

Signature: `addMove(self, col, ox)`

This method takes two inputs: the first input `col` represents the index of the column to which the checker will be added; the second input `ox` will be a 1-character string representing the checker to add to the board. That is, `ox` should either be `'X'` or `'O'` (again, capital O, not zero).

Remember that the checker slides down from the top of the board! Thus, your code will have to find the appropriate row number available in column `col` and put the checker in that row. In `addMove` you do **not** have to check that `col` is a legal column number or that there is space in column `col`. That checking is important, however. The next method, which is called `allowsMove`, will do just that.

Testing addMove

Here is a sequence for testing `addMove`—try it out!

```
b = Board(7,6)
b.addMove(0, 'X')
b.addMove(0, 'O')
b.addMove(0, 'X')
b.addMove(3, 'O')
b.addMove(4, 'O')  # cheating by letting O go
again!
b.addMove(5, 'O')
b.addMove(6, 'O')
print b

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

`clear( self )`

`clear( self )` should clear the board that calls it.

Not much to say about `clear( self )`. It's useful, though!

`setBoard`, for applying *many* moves to a board

This one is really useful for quickly creating a board to test your `winsFor` in the next part. Here is the code we used for `setBoard`—you should include this in your `Board` class, because it makes testing much easier!

```
    def setBoard( self, moveString ):
        """ takes in a string of columns and
places
            alternating checkers in those
columns,
            starting with 'X'

            For example, call
b.setBoard('012345')
            to see 'X's and 'O's alternate on the
            bottom row, or b.setBoard('000000')
to
            see them alternate in the left
column.

            moveString must be a string of
integers
        """
        nextCh = 'X'   # start by playing 'X'
        for colString in moveString:
            col = int(colString)
            if 0 <= col <= self.width:
                self.addMove(col, nextCh)
            if nextCh == 'X': nextCh = 'O'
            else: nextCh = 'X'
```

`allowsMove`, for checking if a column is a legal move

Signature: `allowsMove(self, c)`

This method should return `True` if the calling object (of type `Board`) **does** allow a move into column `c`. It returns `False` if column `c` is not a legal column number for the calling object. It also returns `False` if column `c` is full. Thus, this method should check to be sure that `c` is within the range from 0 to the last column *and* make sure that there is still room left in the column!

Testing `allowsMove`

Here is an example sequence for testing—try it!

```
>>> b = Board(2,2)
>>> b

| | |
| | |
-----
 0 1

>>> b.addMove(0,
'X')
>>> b.addMove(0,
'O')
>>> b

|O| |
|X| |
-----
 0 1

>>> b.allowsMove(-1)
False

>>> b.allowsMove(0)
False

>>> b.allowsMove(1)
True

>>> b.allowsMove(2)
False
```

`isFull`, for checking if the board is full

Signature: `isFull(self)`

This method should return `True` if the calling object (of type `Board`) is completely full of checkers. It should return `False` otherwise. Notice that you can leverage `allowsMove` ot make this method very concise! Unless you're supernaturally patient, you'll want to test this on small boards.

Testing `isFull`

Here is an example sequence for testing (it uses `setBoard`, above)

```
>>> b = Board(2,2)
>>> b.isFull()
False

>>> b.setBoard( '0011'
)
>>> b

|O|O|
|X|X|
-----
 0 1

>>> b.isFull()
True
```

`delMove`, for removing a checker from the board

Signature: `delMove(self, c)`

This method should do the opposite of `addMove`. It should remove the top checker from the column `c`. If the column is empty, then `delMove` should do nothing. This function may not seem useful now, but it will become *very* useful when you try to implement your own Connect Four player.

Testing `delMove`

Here is an example sequence for testing:

```
>>> b = Board(2,2)
>>> b.setBoard( '0011'
)
>>> b.delMove(1)
>>> b.delMove(1)
>>> b.delMove(1)
>>> b.delMove(0)

>>> b

| | |
|X| |
-----
 0 1
```

`winsFor`, checks if someone has won the game

Signature: `winsFor(self, ox)`

This method's input `ox` is a 1-character checker: either `'X'` or `'O'`. It should return `True` if there are four checkers of type `ox` in a row on the board. It should return `False` othwerwise.

Important Note: you need to check if the player has won horizontally, vertically, or diagonally (and there are two different directions for a diagonal win).

One way to approach this is to consider each possible *anchor* checker that might start a four-in-a-row run. for example, all of the "anchors" that might start a horizontal run (going from left to right) must be in the columns *at least four places from the end of the board*. That constraint will help you avoid out-of-bounds errors. Here is some starter code that illustrates this technique (but as of yet only checks for horizontal wins):

```
        H = self.height
        W = self.width
        D = self.data
        # check for horizontal wins
        for row in range(0,H):
            for col in range(0,W-3):
                if D[row][col] == ox and 
\
                   D[row][col+1] == ox 
and \
                   D[row][col+2] == ox 
and \
                   D[row][col+3] == ox:
                    return True
```

Note the backslash characters—these tell Python that the line of code will continue on the next line of the file.

Note, too, the `-3` that keeps the checking in bounds. Different directions will require different guards against going out of bounds.

Warning

It's better **not** to explicitly *count* checkers to see if you reach four. The problem is that you must visit each checker in the right order. Vertical and horizontal orderings aren't bad, but visiting each checker in diagonal order is neither easy nor informative. It's more convenient to check for all four checkers at once, as in the previous example.

Testing `winsFor`

This is an important method to test thoroughly! Here is an example sequence for testing:

```
>>> b = Board(7,6)
>>> b.setBoard( '00102030' )
>>> b.winsFor('X')
True

>>> b.winsFor('O')
True


>>> b = Board(7,6)
>>> b.setBoard( '00102030' )
>>> b.winsFor('X')
True

>>> b.winsFor('O')
True

>>> b = Board(7,6)
>>> b.setBoard( '23344545515'
)
>>> b

| | | | | | | |
| | | | | | | |
| | | | | |X| |
| | | | |X|X| |
| | | |X|X|O| |
| |O|X|O|O|O| |
---------------
 0 1 2 3 4 5 6

>>> b.winsFor('X')   # diagonal
True

>>> b.winsFor('O')
False
```

`hostGame`, hosts a full game of Connect Four

Signature: `hostGame(self)`

This method brings everything together into the familiar game. It should host a game of connect four, using the methods listed above to do so. In particular, it should alternate turns between `'X'` (who will always go first) and `'O'` (who will always go second). It should ask the user (with the `input` function) to select a column number for each move. See below for an example of how the interaction should work, but here are a few important points to keep in mind:

- This method should print the board before prompting for each move.
- You will probably want to use a large `while` loop to structure the game. You should have `'X'` go first and `'O'` go second. My suggestion would be to put both `'X'`s and `'O'`s turn into the body of the `while` loop. Thus, one iteration of the `while` loop will make two connect-four turns.
- You might also use the infinite loop `while True:` and then use `break` when the game ends somewhere in

the loop's body.

- After each `input`, you should check if the column chosen is a valid one. Thus, this method should detect illegal moves, either out-of-bounds or a full column, and prompt for another move instead. You do not have to check if the input is an integer, however; you may assume it will always be one. So, do use `input` instead of `raw_input`.
- As a guide to how you might handle the case when a user inputs an *incorrect* move, consider the following small loop:

```
users_col = -1
while self.allowsMove( users_col ) == False:
    users_col = input("Choose a column: ")
```

  The above code will simply continue to prompt the user for a column number that is valid until it receives one.
- This `hostGame` method should place each checker into the user's chosen (valid!) column. Then, it should check if that player has won the game or if the board is now full.
- If the game is over for either reason, the game should stop, the board should be printed out one last time, and the program should report who won (or that it was a tie.) Note that you can use `break` to get out of a loop—even if that `break` is within an if/else statement.
- If the game is not over, the other player should be prompted to make a move, and so on.

Be sure to test this method by playing the game a few times (with each possible ending)!

Here is an example run, to give a sense of the input and output:

```
>>> b = Board(7,6)
>>> b.hostGame()


Welcome to Connect
Four!

| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
---------------
 0 1 2 3 4 5 6

X's choice:  3

| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | |X| | | |
---------------
 0 1 2 3 4 5 6

O's choice:  4
```

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

X's choice:  2

| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | |X|X|O| | |
---------------
 0 1 2 3 4 5 6

O's choice:  4

| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | |O| | |
| | |X|X|O| | |
---------------
 0 1 2 3 4 5 6

X's choice:  1

| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | |O| | |
| |X|X|X|O| | |
---------------
 0 1 2 3 4 5 6

O's choice:  2

| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | |O| |O| | |
| |X|X|X|O| | |
---------------
 0 1 2 3 4 5 6

X's choice:  0
```

```
X wins—Congratulations!

| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | |O| |O| | |
|X|X|X|X|O| | |
---------------
 0 1 2 3 4 5 6

>>>
```

Congratulations! You've finished the Connect Four `Board` class.

Next week's assignment will give you the chance to extend this connect-four board and build an "AI" player for the game.

Submit Homework 10, Problem 2

30.0/30.0 points (graded)

To submit your Homework 10, Problem 2 code, you'll need to copy it from the trinket above and paste it into the box below. After you've pasted your code below, click the "Check" button.

IMPORTANT: Make sure that there aren't spaces at the beginning of your code, and that you copied all of the characters. If there are extra spaces or you are missing spaces, our server won't be able to run your code and we won't be able to give you any of the points you deserve for your hard work.

```
class Board:
    """ a datatype representing a C4 board
        with an arbitrary number of rows and cols
    """

    def __init__( self, width, height ):
        """ the constructor for objects of type Board """
        self.width = width
        self.height = height
        W = self.width
        H = self.height
        self.data = [ [' ']*W for row in range(H) ]



    def __repr__(self):
        """ this method returns a string representation
            for an object of type Board
        """
```

25

```
        H =
self.height
```

Press ESC then TAB or click outside of the code editor to exit
correct

correct

Test results
CORRECT See full outputSee full output

You have used 1 of 3 attempts Some problems have options such as save, reset, hints, or show answer. These options follow the Submit button.