Problem 1: The Connect Four Player Class

This problem asks you to write another connect-four-related class named `Player`. You should start by making a copy of your Homework 10, Problem 2 trinket.

In this problem, you will need to create a class named `Player` that evaluates connect-four boards and decides where to move next. The basic approach is the following:

- Look at each column on the board. Give each column a numeric score:
  - **-1.0** represents a column that is full (so no move can be made there)
  - **0.0** represents a column that, if chosen as the next move, will result in a **loss** for the the player.
  - **50.0** represents a column that, if chosen as the next move, will not produce a win nor produce a loss for the player (at least not in the near-term future)
  - **100.0** represents a column that, if chosen as the next move, will result in a **win** for the the player.
- After obtaining a list of scores in the above format, one score per column, the computer player will choose a move by finding the column with the maximum score and playing there. If there are ties (and there will be), one of these tie-breaking strategies is used:
  - **LEFT**: pick the leftmost high score
  - **RIGHT**: pick the rightmost high score
  - **RANDOM**: pick one of the high scores randomly

The more detailed descriptions below will provide a skeleton and a couple of hints for the design of your `Player` class and how to test it.

The `Player` Class

Your `Player` class should have at least these three data members:

- A one-character string representing the checker, either `'X'` or `'O'`, being used by the connect-four `Player`. **Warning!** Remember that `'O'` is capital-o, not zero. One reasonable name for this data member might be `self.ox`.
- A string, either `'LEFT'`, `'RIGHT'`, or `'RANDOM'`, representing the *tiebreaking type* of the player. This is the name for one of the three strategies described above. One reasonable name for this data member might be `self.tbt` (for *tiebreaking type*).
- A nonnegative integer representing how many moves into the future the player will look in order to evaluate possible moves. One reasonable name for this data member might be `self.ply` because one turn of gameplay is sometimes called a *ply*.

You should provide your `Player` class with write the following methods. Be sure to try the hints on how to test each

one after writing it!

`__init__`

Signature: `__init__(self, ox, tbt, ply)`:

This is a constructor for `Player` objects that takes three arguments. (Remember that `self` refers to the object being constructed and that it is not explicitly passed into the constructor.)

- This constructor first takes in a one-character string `ox`: this will be either `'X'` or `'O'`.
- Second, it takes in `tbt`, a string representing the tiebreaking type of the player. It will be one of `'LEFT'`, `'RIGHT'`, or `'RANDOM'`.
- The third input `ply` will be a nonnegative integer representing the number of moves that the player should look into the future when evaluating where to go next.

Inside the constructor, you should set the values of the data members of the object. There's not much else to do. (See below for help on this code!)

`__repr__`

Signature: `__repr__(self)`

This method returns a string representing the `Player` object that calls it. This should simply print the three important characteristics of the object: its checker string, its tiebreaking type, and its ply. Since we went over these two methods in class, feel free to use that code (provided here):

```
class Player:
    """ an AI player for Connect Four """

    def __init__( self, ox, tbt, ply ):
        """ the constructor """
        self.ox = ox
        self.tbt = tbt
        self.ply = ply

    def __repr__( self ):
        """ creates an appropriate string """
        s = "Player for " + self.ox + "\n"
        s += "  with tiebreak type: " + self.tbt +
"\n"
        s += "  and ply == " + str(self.ply) + "\n\n"
        return s
```

Testing `__init__` and `__repr__`

```
>>> p = Player('X', 'LEFT', 2)
>>> p
Player for X
  with tiebreak: LEFT
  and ply == 2

>>> p = Player('O', 'RANDOM',
0)
>>> p
Player for O
  with tiebreak: RANDOM
  and ply == 0
```

Admittedly, testing at this point is mostly to familiarize yourself with objects of type `Player`.

`oppCh`

Signature: `oppCh(self)`

This method should return the **other** kind of checker or playing piece, i.e., the piece being played by `self`'s opponent. In particular, if `self` is playing `'X'`, this method returns `'O'` and vice-versa. Just be sure to stick with capital-`O`! This method is easy to test:

```
>>> p = Player('X', 'LEFT', 3)
>>> p.oppCh()
'O'
>>> Player('O', 'LEFT',
0).oppCh()
'X'
```

`scoreBoard`

Signature: `scoreBoard(self, b)`

This method should return a *single* `float` value representing the score of the input `b`, which you may assume will be an object of type `Board`. This should return `100.0` if the board `b` is a win for `self`. It should return `50.0` if it is neither a win nor a loss for `self`, and it should return `0.0` if it is a loss for `self` (i.e., the opponent won).

Testing `scoreBoard`

You should test all three possible output scores—here is an example of how to test the first case. For easy copy-and-paste, many statements are in one large line:

```
>>> b = Board(7,6)
>>> b.setBoard( '01020305' )
>>> b

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

>>> p = Player( 'X', 'LEFT', 0 )
>>> p.scoreBoard(b)
100.0

>>> Player('O', 'LEFT', 0).scoreBoard(b)
0.0

>>> Player('O', 'LEFT', 0).scoreBoard( Board(7,6)
)
50.0
```

The last two examples are using objects that have not yet been assigned variable names.

Note that the tiebreak type does not affect this method at all. You can save time by having `scoreBoard` use the `winsFor` method. Recall that `winsFor` is in the `Board` class, however. It's likely that the `oppCh` method will help here!

`tiebreakMove`

Signature: `tiebreakMove(self, scores)`

This method takes in `scores`, which will be a nonempty list of floating-point numbers. **If there is only one highest score** in that `scores` list, this method should return **its column number**, not the actual score! Note that the column number is the same as the index into the list `scores`. If there is *more than one* highest score because of a tie, this method should return the **column number** of the highest score appropriate to the player's tiebreaking type.

Thus, if the tiebreaking type is `'LEFT'`, then `tiebreakMove` should return the **column** of the leftmost highest score (not the score itself). If the tiebreaking type is `'RIGHT'`, then `tiebreakMove` should return the **column** of the rightmost highest score (not the score itself). And if the tiebreaking type is `'RANDOM'`, then `tiebreakMove` should return the **column** of the a randomly-chosen highest score (yet again, not the score itself).

One possible way to write this method is to *first create a list of **indices** at which* `scores` *contains its max element*. For example, if `scores` consisted of `[50,50,50,50,50,50,50]`, then `maxIndices` would be `[0,1,2,3,4,5,6]`. On the other hand, if `scores` was the list `[50,100,100,50,50,100,50]`, then `maxIndices` would be `[1,2,5]` .

Testing `tiebreakMove`

You should test for all three tiebreaking types. Here are two tests:

```
>>> scores = [0, 0, 50, 0, 50, 50,
0]
>>> p = Player('X', 'LEFT', 1)
>>> p2 = Player('X', 'RIGHT', 1)
>>> p.tiebreakMove(scores)
2
>>> p2.tiebreakMove(scores)
5
```

A hint on `tiebreakMove`: it's helpful to find the max of the list first (with Python's built-in `max` function) and then search for the column in which the max is located—*starting from an appropriate initial position*—and the `self.tbt` string is what determines the appropriate initial position from which to start that search.

`scoresFor`

Signature: `scoresFor(self, b)`

This method is the heart of the `Player` class! Its job is to return a list of scores, with the `c`th score representing the "goodness" of the input board *after the player moves to column* `c`. And, "goodness" is measured by what happens in the game after `self.ply` moves.

Admittedly, that is a lot to handle! So, here is a breakdown:

1. First, this method creates a list of all fifties (I called it `scores`) with length equal to the number of columns in the board `b`. Remember that you can use list multiplication: `[50]*b.width`.
2. Then, this method loops over all possible columns.
3. **Base Case** If a particular column is full, it assigns a `-1.0` score for that column.
4. **Another Base Case** If the game has *already* been won for `self` or for the opponent, then there's no point in making any additional moves (indeed, it's not allowed!)—simply evaluate the board and use that score for the current column under consideration.
5. **And Another Base Case** Finally the *real* base case! If the object's `ply` is `0`, no move is made. What's more, if you've handled the column-full and game-over cases above, there's only one score left to give the column with a 0-ply lookahead what is it? This is to be expected if the player is not looking at all into the future.
6. **Recursive Case** But, if the object's ply is greater than `0` and the game isn't over, **the code should make a move into the column that is under consideration**. This will use some of the methods in `Board`. In addition, it's a good idea to check if the new, resulting board has ended the game if so, give the column the appropriate score.
7. Once that move is made, if the game is *not* over, the crucial task is to determine **what scores an opponent would give the resulting board**.
8. This means creating an opponent (which will be of `Player` class!) It's safe to assign this to-be-constructed opponent player the same tiebreaking-style as you do—that is, the same as `self`. Once you've created an opposing player, you should use recursion to determine the seven scores it would give to its next move.
9. However, the scores reported by the opponent are NOT the scores that `self` should use: after all, the opponent has a different goal than `self`! Rather, you will want to compute `self`'s evaluation of the board based on the list of opponent's scores. Then, assign the resulting score to the value of the current column's move.
10. Be sure to **delete** the checker that had been placed throughout the evaluation of this particular column!

11. Once all of the possible moves have been evaluated, the `scoresFor` method should return the complete list of scores, one per column. So, in a seven-column board, there will be seven numbers in the list returned.

Testing `scoresFor`

Here is a case that will test almost all of your `scoresFor` method, the commands to set up the board have been placed on one line for easy copy-and-paste into your Python window:

```
>>> b = Board(7,6)
>>> b.setBoard( '1211244445' )
>>> b

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

# 0-ply lookahead doesn't see threats
>>> Player('X', 'LEFT', 0).scoresFor(b)
[50.0, 50.0, 50.0, 50.0, 50.0, 50.0, 50.0]

# 1-ply lookahead sees immediate wins
# (if only it were 'O's turn!)
>>> Player('O', 'LEFT', 1).scoresFor(b)
[50.0, 50.0, 50.0, 100.0, 50.0, 50.0,
50.0]

# 2-ply lookahead sees possible losses
# ('X' better go to column 3)
>>> Player('X', 'LEFT', 2).scoresFor(b)
[0.0, 0.0, 0.0, 50.0, 0.0, 0.0, 0.0]

# 3-ply lookahead sees set-up wins
# ('X' sees that col 3 is a win!)
>>> Player('X', 'LEFT', 3).scoresFor(b)
[0.0, 0.0, 0.0, 100.0, 0.0, 0.0, 0.0]

# At 3-ply, 'O' does not see any danger
#  if it moves to columns on either side
>>> Player('O', 'LEFT', 3).scoresFor(b)
[50.0, 50.0, 50.0, 100.0, 50.0, 50.0,
50.0]

# But at 4-ply, 'O' does see the danger!
# again, too bad it's not 'O's turn
>>> Player('O', 'LEFT', 4).scoresFor(b)
[0.0, 0.0, 0.0, 100.0, 0.0, 0.0, 0.0]
```

This last test may take a few seconds, even on a fast computer.

`nextMove`

Signature: `nextMove(self, b)`

This method takes in `b`, an object of type `Board` and returns an integer—namely, the column number that the calling object (of class `Player`) chooses to move to. This is the primary interface to `Player`, but it is really just a "wrapper" for the heavy lifting done by the other methods, particularly `scoresFor`. Thus, `nextMove` should use `scoresFor` and `tiebreakMove` to return its move.

Testing `nextMove`

This is similar to the previous example.

```
>>> b = Board(7,6);
>>> b.setBoard( '1211244445' )
>>> b

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

>>> Player('X', 'LEFT', 1).nextMove(b)
0

>>> Player('X', 'RIGHT', 1).nextMove(b)
6

>>> Player('X', 'LEFT', 2).nextMove(b)
3

# the tiebreak does not matter
# if there is only one best move
>>> Player('X', 'RIGHT', 2).nextMove(b)
3

# again, the tiebreak does not matter
# if there is only one best move
>>> Player('X', 'RANDOM',
2).nextMove(b)
3
```

Putting it all together: `Board`'s `playGame` method

Add the following method **to your** `Board` **class**.

Signature: `playGame(self, px, po)`

Copy and change your `hostGame` method from Homework 10, Problem 2 in order to create this `playGame` method.

`playGame` does just that—it calls on the `nextMove` method in `px` and `po`, which will be objects of type `Player` in order to play a game.

One additional twist: you should handle the case in which either `px` or `po` is the string `'human'` instead of an object of type `Player`. In this `'human'` case, the `playGame` method should simply pause and ask the user to input the next column to move for that player, with error-checking just as in `hostGame`.

See below for some determininstic—that is, non-random—examples that you can use to test your `playGame` method:

Testing `playGame`

Some deterministic examples:

```
>>> px = Player('X', 'LEFT',
0)
>>> po = Player('O', 'LEFT',
0)
>>> b = Board(7,6)
>>> b.playGame(px, po)

# Lots of boards omitted

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

X wins!
```

Example #2 (notice the game ends faster!):

```
>>> px = Player('X', 'LEFT',
1)
>>> po = Player('O', 'LEFT',
1)
>>> b = Board(7,6)
>>> b.playGame(px, po)

# Lots of boards omitted

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

X wins!
```

Example #3 (the higher ply does not always win!):

```
>>> px = Player('X', 'LEFT',
3)
>>> po = Player('O', 'LEFT',
2)
>>> b = Board(7,6)
>>> b.playGame(px, po)

# Lots of boards omitted

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

O wins!
```

That's it You've built your Connect Four AI!

Submit Homework 11, Problem 1

65.0/65.0 points (graded)

To submit your Homework 11, Problem 1 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.

```
1
2
3
4
5
6 import random
7
8 class Board:
9     """ a datatype representing a C4
board
10         with an arbitrary number of rows and
cols
11
```

```
"""
12
13
    def __init__(self, width,
height):
14
        """ the constructor for objects of type Board
"""
15
        self.width =
width
16
        self.height =
height
17
        W =
self.width
18
        H =
self.height
19
        self.data = [ [' ']*W for row in range(H)
]
20
```

```
21



22

    def
__repr__(self):

23

        """ this method returns a string
representation

24

            for an object of type
Board

25

"""
```

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.