Problem 2, Part 2: Connect Four: The `Player` class [40 points; individual or pair]

Copied from: https://www.cs.hmc.edu/twiki/bin/view/CS5/Connect4Ply on 3/15/17

Please include this `Player` class in the same `hw11pr2.py` file that has the previous part's `Board` class.

The `Player` class—a preview

This problem asks you to write another Connect-Four-related class named `Player`. Your `Player` class will evaluate Connect-Four boards and decide where to move next. The basic approach is as follows:

- 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 produce neither a win nor a loss for the player (at least not in the near future…)
  - **100.0** represents a column that, if chosen as the next move, will result in a **win** for 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 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.tiebreak`.

- A nonnegative integer representing how many moves into the future the player will look to evaluate possible moves. One reasonable name for this data member might be `self.ply` because one turn of game play is sometimes called a *ply*. The other legal value for `self.ply` is the string **'HUMAN'**, which will indicate that the object represents a human player. (It will ask for a move from the user in this case.)

Methods required for the `Player` class :

__init__

- `__init__(self, ox, tiebreak, 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.) The constructor first takes in a one-character string `ox`: this will be either `'X'` or `'O'`. Second, it takes in `tiebreak`, a string representing the tiebreaking type of the player. It will be one of `'LEFT'`, `'RIGHT'`, or `'RANDOM'`. The third argument, `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. The argument `ply` may also be the string `'HUMAN'`, meaning that the object will represent a human player. (It will ask for a move from the user in this case.)

Inside the constructor, you should set the values of the data members of the object. There's not much else to do!

__repr__

- `__repr__(self)`:

  This method returns a string representing the `Player` object that calls it. The string should simply represent the three important characteristics of the object: its checker string, its tiebreaking type, and its ply.

Testing ***init*** and `__repr__`:

```
In [1]: p = Player('X', 'LEFT', 2)

In [2]: p
Out[2]: Player for X
  with tiebreak: LEFT
  and ply == 2

In [3]: p = Player('O', 'RANDOM', 'HUMAN')

In [4]: p
Out[4]: Player for O
  with tiebreak: RANDOM
  and ply == HUMAN
```

You might wonder why an object of type `Player` even needs a tiebreaking type if it is of `'HUMAN'` ply… It really doesn't—the tiebreaking type will be ignored in that case.

oppCh

- `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:

```
In [1]: p = Player('X', 'LEFT', 3)

In [2]: p.oppCh()
Out[2]: 'O'

In [3]: Player('O', 'LEFT', 0).oppCh()
Out[3]: 'X'
```

scoreBoard

- `scoreBoard(self, b)`:

  This method should return a *single* `float` value representing the score of the argument `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:

```
In [1]: b = Board(7,6)  # must be width, height

In [2]: b.addMove(0,'X')

In [3]: b.addMove(0,'X')

In [4]: b.addMove(0,'X')

In [5]: b.addMove(0,'X')

In [6]: p = Player('X', 'LEFT', 0)

In [7]: b             # just to check...

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

In [8]: p.scoreBoard(b)
Out[6]: 100.0
```

tiebreakMove

- `tiebreakMove(self, scores)`:

  This method accepts `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).

Testing tiebreakMove

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

```
In [1]: scores = [0, 0, 50, 0, 50, 50, 0]

In [2]: p = Player('X', 'LEFT', 1)

In [3]: p2 = Player('X', 'RIGHT', 1)

In [4]: p.tiebreakMove(scores)
Out[4]: 2

In [5]: p2.tiebreakMove(scores)
Out[5]: 5
```

scoresFor

- `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 board given as `b` *after the player moves to column* `c`. To make the program play well, "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. The method first creates a list of all zeros, with length equal to the number of columns in the board `b`.
2. It then assigns a **-1** for each column that is full.
3. If the `Player` object's ply is 0, no moves are made, and the scores should all be set "appropriately"—be careful, because this can involve several cases, depending on whether the game is already over or not—and the order in which you check matters!
4. But if the `Player` object's ply is greater than 0 and the game isn't over, it makes a trial move into each of the non-full columns of the board (using the methods in `Board`). For each column, it figures out what scores an opponent would give the resulting board using `ply-1` and computes its final score based on the opponent's evaluation. **To do this,** the easiest thing is to make a new opponent (an object from this same `Player` class!!) whose `ox` is the opposite of yours and whose `ply` is one less than yours. Let this opponent recursively score the board. Your score for your choice of move will now be computed simply from the opponents score. Finally, after you've done this for every possible move that you could make, you return the complete list of scores. While this function is the "brains" of the entire program, it's actually quite short. (Our sample implementation is only about 20 lines long.)

Testing scoresFor

Here is a case that will test almost all of your `scoresFor` method.
The `>>>` prompt has been removed at the top for easy copy-and-paste into your Python window:

```
b = Board(7,6)    # must be width, height
b.setBoard('1211244445')

In [1]: 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...
In [1]: p = Player('X', 'RIGHT', 0)

In [2]: p.scoresFor(b)
Out[2]: [50.0, 50.0, 50.0, 50.0, 50.0, 50.0, 50.0]

# The tiebreak method doesn't matter...
In [1]: p = Player('X', 'LEFT', 0)

In [2]: p.scoresFor(b)
Out[2]: [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!)
In [1]: p = Player('O', 'LEFT', 1)

In [2]: p.scoresFor(b)
Out[2]: [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...)
In [1]: p = Player('X', 'LEFT', 2)

In [2]: p.scoresFor(b)
Out[2]: [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!)
In [1]: p = Player('X', 'LEFT', 3)

In [2]: p.scoresFor(b)
Out[2]: [0.0, 0.0, 0.0, 100.0, 0.0, 0.0, 0.0]

# At 3-ply, 'O' does not see any danger
In [1]: p = Player('O', 'LEFT', 3)

In [2]: p.scoresFor(b)
Out[2]: [50.0, 50.0, 50.0, 100.0, 50.0, 50.0, 50.0]
```

```
# At 4-ply, 'O' does see the danger!
# again, too bad it's not 'O's turn...
In [1]: p = Player('O', 'LEFT', 4)

In [2]: p.scoresFor(b)
Out[2]: [0.0, 0.0, 0.0, 100.0, 0.0, 0.0, 0.0]
```

nextMove

- `nextMove(self, b)`:

  This method accepts `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, if the `self.ply` data member is numeric (and thus not `'HUMAN'`), then `nextMove` should use `scoresFor` and `tiebreakMove` to return its move. On the other hand, if the ply *is* `'HUMAN'`, then this method should simply ask the user to enter a move. In this latter case, it is simply acting as a software place-holder.

Testing nextMove

This continues the previous example…again, the `>>>` have been removed at the top for easy copy-and-paste into your Python window:

```
b = Board(7,6)    # must be width, height!!
b.setBoard('1211244445')

In [1]: b

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

In [2]: p = Player('X', 'LEFT', 1)
```

```
In [3]: p.nextMove(b)
Out[3]: 0

In [4]: p = Player('X', 'RIGHT', 1)

In [5]: p.nextMove(b)
Out[5]: 6

In [6]: p = Player('X', 'LEFT', 2)

In [7]: p.nextMove(b)
Out[7]: 3

# the tiebreak does not matter
# if there is only one best move...
In [1]: p = Player('X', 'RIGHT', 2)

In [2]: p.nextMove(b)
Out[2]: 3

# again, the tiebreak does not matter
# if there is only one best move...
In [1]: p = Player('X', 'RANDOM', 2)

In [2]: p.nextMove(b)
Out[2]: 3

# human players are interactive
In [1]: p = Player('X', 'LEFT', 'HUMAN')

In [2]: p.nextMove(b)
Where would you like to place a checker? 2
Out[2]: 2
```

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

playGame

Add the following method **to your** `Board` **class** in hw11pr2.py !

- `playGame(self, p1, p2)`: This method wraps everything into a game of Connect Four—this time, using computer players of type `Player`! This `playGame` method takes in two objects, `p1` and `p2`, of type `Player`. It then alternates their moves (calling `nextMove`) in the usual way, printing out the updated board after each move. Note that if one (or both) of the computer players has ply equal to `'HUMAN'`, the game will pause and prompt the user to choose a move for that player, as was described in `nextMove`. If both are pure computer players, however, they will quickly finish their game. See below for a deterministic (non-random) example that you can use to test your `playGame` method:

Testing playGame

A deterministic example:

```
In [1]: p2 = Player('O', 'LEFT', 0)

In [2]: p1 = Player('X', 'LEFT', 0)

In [3]: b = Board(7,6)    # must be width, height!

In [4]: b.playGame(p1,p2)

# 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!
```

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

Please submit this file as `hw11pr2.py` to the submission Page in the usual way.