Homework 7

Due 4:00pm, Friday, March 22, 2019

Installation and Handin

Homework Installation. To copy support files to your to your home directory for this homework type the following in a Brown CS terminal window:

```
cs4_install hw07
```

There should now be a `hw07` folder within your homeworks directory. Using Terminal, you can move into the `hw07` folder with the `cd` command:

```
cd ~/course/cs004/homeworks/hw07
```

Homework hand-in. Be sure to turn in all the files requested and that they are named exactly as specified, including spelling and case. When you're ready to submit the files, run:

```
cs4_handin hw07
```

from a Brown CS Terminal window from your `~/course/cs004/homeworks/hw07` directory. The entire contents of `~/course/cs004/homeworks/hw07` will be handed in. Check for a confirmation email to ensure that your assignment was correctly submitted using the `cs4_handin` command. You can resubmit this assignment using the `cs4_handin` command at any time, but be careful, as only your most recent submission with be graded.

Harder parts of this problem set are indicated by *. Problems that may be challenging are also indicated by * following their name.

Note: Be sure to read each section all the way through before starting work on that problem, and make sure to write your test cases *before* starting to write code.

Part I: Connect Four Setup (25)

Include your answers and code to this part of the homework in `hw07_1.py`

Silly Prelude

The babies and puppies are now in a civil disagreement on which is the smarter species. You know that the species you side with is obviously superior, so you challenge the other side to a game of Connect Four!

How to Play

The game is played by two players, and the goal is to place four checkers in a line vertically, horizontally, or diagonally. The players alternate turns and add one checker to the board at a time. However, because the board stands vertically, a checker cannot be placed in an arbitrary position on the board. Rather, a checker must be inserted at the top of one of the columns, and it drops down as far as it can go – until it rests on top of the existing checkers in that column, or (if it is the first checker in that column) until it reaches the bottom row of the column. The standard board size for Connect Four is six rows by seven columns.

In order to represent the Connect Four board in the computer we have given you a `Board` class defined in the file **`board.py`**. Below is a complete list of all the classes' methods and what they do. If you would like to know *how* the methods work, you should take a read through **`board.py`**!

Constructor	What It Does
`Board(height, width)`	Initializes an empty Board object with `height` rows and `width` columns.

Properties	What It Is
`height`	the number of rows the board has
`width`	the number of columns the board has
`slots`	a 2D list containing what is being held in each slot of the board (`'X'`, `'O'`, or `' '`). To access a slot in row `i` and column `j`, use `board.slots[i][j]`. Note: `board.slots[0][0]` represents the top left corner of the board, and `board.slots[5][6]` represents the bottom right corner.

method name	what it does
`__repr__()`	returns a string version of the board for printing
`add_checker(checker, col)`	adds the `checker` to the board in column `col`
`reset()`	clears the board
`add_checkers(col_nums)`	adds a checker to the board for each number in the string `col_nums`, for example `board.add_checkers('1123')` will add checkers, starting with `'X'` and alternating, to column 1 then 1 then 2 then 3.
`can_add_to(col)`	returns a Boolean representing whether or not you can add a checker to column `col`
`is_full()`	returns a Boolean representing whether the board is completely full or not
`remove_checker(col)`	removes the last placed checker in column `col` if it exists
`is_checker(row, col, checker)`	returns `True` if the checker at row `row` and column `col` is equal to `checker` and `False` otherwise
`is_win_for(checker)`	returns `True` if the board is a win for `checker` and `False` otherwise

Problem 7.1: A Connect Four `Player` Class

In this problem, you will create a `Player` class to represent a player of the Connect Four game that is compatible with the `connect_four` play loop.

Getting started
We have included an `import` statement at the top of `hw07_1.py` that imports the `Board` class from the `board.py` file. Therefore, you will be able to use `Board` objects and their methods as needed.

1. Write a constructor `__init__(self, checker)` that constructs a new `Player` object by initializing the following two attributes:
    - an attribute `checker` – a one-character string that represents the gamepiece for the player (either an `'X'` or `'O'`), as specified by the parameter `checker`
    - an attribute `num_moves` – an integer that stores how many moves this specific player has made so far. This attribute should be initialized to zero to signify that the `Player` object has not yet made any Connect Four moves.

2. Write a method `__repr__(self)` that returns a string representing a `Player` object. The string returned should indicate which checker the `Player` object is using. For example:

```
>>> p1 = Player('X')
>>> p1
Player X
```

   The results of your `__repr__` method should exactly match the results shown above. Remember that your `__repr__` method should ***return*** a string. It should ***not*** do any printing.

3. Write a method `opponent_checker(self)` that returns a one-character string representing the checker of the `Player` object's opponent. The method may assume that the calling `Player` object has a `checker` attribute that is either `'X'` or `'O'`. For example:

```
>>> p = Player('O')
>>> p.opponent_checker()
'X'
```

4. Write a method named `next_move(self, board)` that accepts a `Board` object as a parameter and returns the column where the player wants to make the next move *without applying that move to the board.* This function should
    - Determine the next move for this version of the Player by asking the user to enter a 0-indexed column number that represents where they want to place a checker on the board. The method should repeatedly ask for a column number until a valid column number is given. A column number is valid if it references a valid column and that column has room for an additional piece in it. Use the `can_add_to` method in the `Board` class to determine this easily.
    - Increment the number of moves that the `Player` object has made.

    In order to get input from the user, you should use the Python `input` function. Because `input` always returns a string, you will need to convert the returned string to an integer to get a column number that you can work with. You may assume that the user types in a string that can be converted to an integer by Python.

Example Output

```
>>> p = Player('X')
>>> b = Board(6, 7)    # valid column numbers are 0 - 6
>>> p.next_move(b)
Enter a column: -1
Try again!
Enter a column: 7
Try again!
Enter a column: 5
5                      # return value of method call
>>> p.num_moves        # number of moves was updated
1
```

Part II: Play Connect Four (30)

Include your answers and code to this part of the homework in `hw07_2.py`

In this part of the homework you'll begin by completing `process_move` so that you can use your `Player` class to play the game against a friend.
In `hw07_2.py` we have provided the `connect_four` game function. It takes in two `Player` objects, and it will be used to run a game of Connect Four between those two `Player`s. As you read over this function, you will see that it takes some preliminary steps, and that it then enters a loop that repeatedly queries each player for their next move and adds it to the board via the `process_move` function.

Problem 7.2a `process_move`

Write a function `process_move(player, board)` that takes two parameters: a `Player` object for the player whose move is being processed, and a `Board` object for the game that is being played.

The function will perform all of the steps involved in processing a single move by the specified `player` on the specified `board`. These steps are enumerated below. ***Note that the function should not be very long, because it should take advantage of the methods in the*** *`Player`* ***object and*** *`Board`* ***object that it has been given.*** Those methods will do almost all of the work for you!

Here are the steps that the function should perform:

1. Print a message that specifies whose turn it is:
   `Player X's turn`
   or
   `Player O's turn`
   Important: You should *not* need an `if` statement here. Simply take advantage of the `__repr__` method in `player` to obtain its string representation.

2. Obtain the player's next move by using the appropriate `Player` method. Store the move (i.e., the selected column number) in a variable.

3. Apply the move to the board by using the appropriate `Board` method.

4. Print a blank line, and then print the board.

5. Check to see if the move resulted in a win or a tie by using the appropriate `Board` methods.

   If it is a win, print a message that looks like this:
   `Player X wins in 8 moves.`
   `Congratulations!`
   and return `True`.

   If it is a tie, print
   `It's a tie!`
   and return `True`.

6. If it is neither a win nor a tie, the method should simply return `False`.

Make sure that the method returns the appropriate value — either *`True`* ***or*** *`False`*.

Example Output

```
>>> b = Board(2, 4)
>>> b.add_checkers('0011223')
>>> b
|O|O|O| |
|X|X|X|X|
---------
 0 1 2 3

>>> process_move(Player('O'), b)
Player O's turn
Enter a column: 3
|O|O|O|O|
|X|X|X|X|
---------
 0 1 2 3

Player O wins in 1 moves.    # we made the other 3 moves for Player O!
Congratulations!
True                         # return value of process_move
>>> b.remove_checker(3)
>>> b.remove_checker(3)      # call this twice!
>>> process_move(Player('O'), b)
Player O's turn
Enter a column: 3

|O|O|O| |
|X|X|X|O|
---------
 0 1 2 3

False
>>> process_move(Player('X'), b)
Player X's turn
Enter a column: 3

|O|O|O|X|
|X|X|X|O|
---------
 0 1 2 3

It's a tie!
True
```

You should also test it from within the context of the `connect_four` function that we have given you. Simply enter the following:

```
>>> connect_four(Player('X'), Player('O'))
```

and then play against a friend, or against yourself! Use Ctrl-C if you need to end the game prematurely.

Problem 7.2b `RandomPlayer`

Define a class called `RandomPlayer` that can be used for an *un*intelligent computer player that chooses at random from the available columns.

This class should be a *subclass* of the `Player` class that you implemented in Part III, and you should take full advantage of inheritance. In particular, you should *not* need to include any attributes in your `RandomPlayer` class, because all of the necessary attributes (the player's checker, and its count of the number of moves) will be inherited from `Player`.

Similarly, you should not need to redefine the `__repr__` or `opponent_checker` methods because they will be inherited from `Player`, and we don't want these methods to behave any differently for a `RandomPlayer` than they do for a `Player`.

However, you will need to do the following:

- Make sure that your class header specifies that `RandomPlayer` inherits from `Player`.
- Write a method `next_move(self, board)` that *overrides* (i.e., replaces) the `next_move` method inherited from `Player`. Rather than asking the user for the next move, this version of `next_move` should choose at random from the columns in the specified `board` that are not yet full, and return the index of that randomly selected column. You may assume that this method will only be called in cases in which there is at least one available column. In addition, make sure that you increment the number of moves that the `RandomPlayer` object has made.

Choosing a Random Move
To ensure that the method does not select the index of a column that is already full, we recommend that you begin by constructing a list containing the indices of all available columns — i.e., all columns to which you can still add a checker. For example, let's say that the parameter `board` represents the following board:

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

The list of available columns in this case would be `[1,2,4,5,6]`.

To build this list, you should consider the columns one at a time, and add the index of any available column to the list. This can be done using a loop or list comprehension. Take advantage of one of the `Board` methods to determine if a given column is available!

- We have included an `import` statement for the `random` module so that you can use the appropriate function to make a random choice from the list of available columns.

Example Output

```
>>> p = RandomPlayer('X')
>>> p
Player X      # uses the inherited __repr__
>>> p.opponent_checker()
'O'           # uses the inherited version of this method
>>> b = Board(2, 4)
>>> b.add_checkers('001223')
>>> b
|O| |X| |
|X|X|O|O|
---------
 0 1 2 3

>>> p.next_move(b)
3             # can be either 1 or 3
>>> p.next_move(b)
1             # can be either 1 or 3
>>> p.next_move(b)
1             # can be either 1 or 3
>>> b.add_checker('O', 1)
>>> b
|O|O|X| |
|X|X|O|O|
---------
 0 1 2 3

>>> p.next_move(b)
3             # must be 3!
>>> p.next_move(b)
3             # must be 3!
```

Playing with the `RandomPlayer` Object

To play against a random player, enter something like this:

```
>>> connect_four(Player('X'), RandomPlayer('O'))
```

You'll see that it's pretty easy to win against someone who chooses randomly!

You could also pit two random players against each other and see who wins:

```
>>> connect_four(RandomPlayer('X'), RandomPlayer('O'))
```

Testing your `RandomPlayer` class
Note: When adding testing functions for your class methods make sure that you define them outside of the class definition itself.

Your tests should verify that:

- `process_move` correctly returns a boolean indicating whether or not the game has ended. Some cases to consider are when Player X wins, Player O wins, when it's a tie, and when the game hasn't reached an endpoint yet.
- `next_move` correctly counts the number of moves a player takes, and that it doesn't try to make moves into columns that are already full/invalid.
    - Optional: Pseudo-random operations, like `random.choice()`, depend on a seed value to determine how they will operate. If you call `random.seed(k)`, where `k` is some specific integer of your choosing, subsequent calls to `random.choice()` will produce the same sequence of "random" choices (hence why these are called *pseudo*-random operations!). As an optional exercise, you can use this fact to generate specific test cases for your `RandomPlayer`'s `next_move` method.

Part III: AI Player (45)*

Include your answers and code to this part of the homework in `hw07_3.py`

You will now define an "intelligent" computer player – one that uses techniques from artificial intelligence (AI) to choose its next move.

In particular, this AI player will *look ahead* some number of moves into the future to assess the impact of each possible move that it could make for its next move, and it will assign a score to each possible move. And since each move corresponds to a column number, it will effectively assign a score to each column.

The possible column scores are:

- -1 for a column that is already full

- 0 for a column that, if chosen as the next move, will result in a *loss* for the player at some point during the number of moves that the player looks ahead.
- 100 for a column that, if chosen as the next move, will result in a *win* for the player at some point during the number of moves that the player looks ahead.
- 50 for a column that, if chosen as the next move, will result in neither a win nor a loss for the player at any point during the number of moves that the player looks ahead.

After obtaining a list of scores for each column, it will choose as its next move the column with the maximum score. This will be the player's judgment of its best possible move.

When looking ahead, the player will assume that its opponent is using a similar strategy – assigning scores to columns based on some degree of lookahead, and choosing what it judges to be the best possible move for itself.

Problem 7.3 `AIPlayer`

Define a class called `AIPlayer` that takes the approach outlined above (and in more detail below) to choose its next move.

Like the `RandomPlayer` class that you implemented for Part III, this class should be a *subclass* of the `Player` class that you implemented in Part II, and you should take full advantage of inheritance.

In addition to the attributes inherited from `Player`, an `AIPlayer` object should include two new attributes:

1. one called `tiebreak` that stores a string specifying the player's tie-breaking strategy (`'LEFT'`, `'RIGHT'`, or `'RANDOM'`)

   If there are ties, the player will use one of the following tie-breaking strategies, each of which is represented by a single-word string and passed into the constructor for `AIPlayer`:
   - `'LEFT'`: out of all the columns that are tied for the highest score, pick the leftmost one. (You should develop a test case for this.)
   - `'RIGHT'`: out of all the columns that are tied for the highest score, pick the rightmost one. (You should develop a test case for this.)
   - `'RANDOM'`: out of all the columns that are tied for the highest score, pick one of them at random. (You can test if the `next_move` is in an allowed set, or you can use the `random.seed` option described previously)

2. one called `lookahead` that stores an integer specifying how many moves the player looks ahead in order to evaluate possible moves.

Getting Started

- Make sure that your class header specifies that `AIPlayer` inherits from `Player`.

- Call the constructor inherited from the superclass, so that it can initialize the inherited attributes:

```
        super().__init__(checker)
```

  This constructor now has two new attributes not inherited from `Player` (see above), so you will need to initialize these by assigning them the values passed in as parameters.

  Make sure that you do *not* redefine the inherited attributes by trying to assign something to them here.
- Write a method `__repr__(self)` that returns a string representing an `AIPlayer` object. This method will override/replace the `__repr__` method that is inherited from `Player`. In addition to indicating which checker the `AIPlayer` object is using, the returned string should also indicate the player's tie-breaking strategy and lookahead.

Example Output

```
>>> p1 = AIPlayer('X', 'LEFT', 1)
>>> p1
Player X (LEFT, 1)
>>> p2 = AIPlayer('O', 'RANDOM', 2)
>>> p2
Player O (RANDOM, 2)
```

The results of your `__repr__` method should exactly match the results shown above.

Writing `max_score_column`

`max_score_column(self, scores)` takes a list `scores` containing a score for each column of the board and returns the *index* of the column with the maximum score. If one or more columns are tied for the maximum score, the method should apply the called `AIPlayer`'s tie breaking strategy to break the tie. Make sure that you return the *index* of the appropriate column, and *not* the column's score.

Notes:

- One good way to implement this method is to first determine the maximum score in `scores` (you can use the built-in `max` function for this), and to then create a list containing the indices of all elements in `scores` that match this maximum score. For example, if `scores` consisted of the list `[50,50,50,50,50,50,50]`, the list of indices that you would build would be `[0,1,2,3,4,5,6]`, because all of these scores are tied for the maximum score. If `scores` consisted of the list `[50,100,100,50,50,100,50]`, you would build the list of indices `[1,2,5]`. Then once you have this list of indices, you can choose from the list based on the `AIPlayer`'s tie breaking strategy.

- If you take this approach, then you don't really need to worry about whether there is a tie. You can *always* use the tie breaking strategy when choosing from the list of indices that you construct!
- We have included an `import` statement for the `random` module so that you can use the appropriate function to make a random choice for players that use the `'RANDOM'` tie breaking strategy.

Examples:

```
>>> scores = [0, 0, 50, 0, 50, 50, 0]
>>> p1 = AIPlayer('X', 'LEFT', 1)
>>> p1.max_score_column(scores)
2
>>> p2 = AIPlayer('X', 'RIGHT', 1)
>>> p2.max_score_column(scores)
5
```

Writing `scores_for`

`scores_for(self, board)` takes a `Board` object, `board`, and determines the `AIPlayer`'s scores for the columns in `board`. Each column should be assigned one of the four possible scores discussed in the start of this problem, based on the called `AIPlayer`'s `lookahead` value. The method should return a list containing one score for each column.
This method should take advantage of both the other methods in the called `AIPlayer`s object (including the inherited ones) *and* the methods in the `Board` object that it is given as a parameter. Don't repeat work that can be done using one of those methods!

You should begin by creating a list (call it `scores`) that is long enough to store a score for each column. You can use list multiplication for this, and it doesn't really matter what initial value you use for the elements of the list.

You should then loop over all of the columns in `board`, determine a score for each column, and assign the score to the appropriate element of `scores`. Here is an outline of the logic:

If the lookahead is 0:

1. If the current column is full, use a score of -1 for it. In other words, assign -1 to the appropriate element of your `scores` list.
2. If `board` is a win for the called `AIPlayer` (i.e., for `self`), use a score of 100 for the current column.
3. If `board` is a win for the player's opponent, use a score of 0 for the current column.
4. Otherwise, use a score of 50 for the column.

If the lookahead is greater than 0:

1. Add one of the called `AIPlayer`'s checkers to the current column using the appropriate `Board` method.
2. Determine what scores *the opponent* would give to the resulting board. To do so, create an opponent (an `AIPlayer` object) with the same tie-breaking strategy as `self`, but with a lookahead that is one less than the one used by `self`. Make a recursive call to determine the scores that this created opponent would give to the current board (the one that resulted from adding a checker to the current column).
    a. Following the approach discussed in lecture, use the opponent's scores (the scores returned by the recursive call) to determine what score `self` should use for the current column.
    b. Remove the checker that was placed in this column so that you can restore `board` to its prior state.

Once the loop has considered all of the columns, the method should return the complete list of scores.

Examples:

```
>>> b = Board(6, 7)
>>> b.add_checkers('1211244445')
>>> b
| | | | | | | |
| | | | | | | |
| | | | |X| | |
| |O| | |O| | |
| |X|X| |X| | |
| |X|O| |O|O| |
---------------
 0 1 2 3 4 5 6
```

```
# A lookahead of 0 doesn't see threats!
>>> AIPlayer('X', 'LEFT', 0).scores_for(b)
[50, 50, 50, 50, 50, 50, 50]
# A lookahead of 1 sees immediate wins.
# (O would win if it put a checker in column 3.)
>>> AIPlayer('O', 'LEFT', 1).scores_for(b)
[50, 50, 50, 100, 50, 50, 50]
# But a lookahead of 1 doesn't see possible losses!
# (X doesn't see that O can win if column 3 is left open.)
>>> AIPlayer('X', 'LEFT', 1).scores_for(b)
[50, 50, 50, 50, 50, 50, 50]
# A lookahead of 2 sees possible losses.
# (All moves by X other than column 3 leave it open to a loss.
# note that X's score for 3 is 50 instead of 100, because it
```

```
        # assumes that O will follow X's move to 3 with its own move
        # to 3, which will block X's possible horizontal win.)
        >>> AIPlayer('X', 'LEFT', 2).scores_for(b)
        [0, 0, 0, 50, 0, 0, 0]
        # A lookahead of 3 sees set-up wins!
        # (If X chooses column 3, O will block its horizontal win, but
        # then X can get a diagonal win by choosing column 3 again!)
        >>> AIPlayer('X', 'LEFT', 3).scores_for(b)
        [0, 0, 0, 100, 0, 0, 0]
        # With a lookahead of 3, O doesn't see the danger of not
        # choosing 3 for its next move (hence the 50s in columns
        # other than column 3).
        >>> AIPlayer('O', 'LEFT', 3).scores_for(b)
        [50, 50, 50, 100, 50, 50, 50]
        # With a lookahead of 4, O **does** see the danger of not
        # choosing 3 for its next move (hence the 0s in columns
        # other than column 3).
```

- Write a method `next_move(self, board)` that *overrides* (i.e., replaces) the `next_move` method that is inherited from `Player`. Rather than asking the user for the next move, this version of `next_move` should return the called `AIPlayer`'s judgment of its best possible move. This method won't need to do much work, because it should use your `scores_for` and `max_score_column` methods to determine the column number that should be returned. In addition, make sure that you increment the number of moves that the `AIPlayer` object has made.

Examples:

```
>>> b = Board(6, 7)
>>> b.add_checkers('1211244445')
>>> b
| | | | | | | |
| | | | | | | |
| | | | |X| | |
| |O| | |O| | |
| |X|X| |X| | |
| |X|O| |O|O| |
---------------
 0 1 2 3 4 5 6

# With a lookahead of 1, gives all columns a score of 50, and its
# tie-breaking strategy leads it to pick the leftmost one.
>>> AIPlayer('X', 'LEFT', 1).next_move(b)
0
# Same lookahead means all columns are still tied, but a different
# tie-breaking strategy that leads it to pick the rightmost column.
```

```
>>> AIPlayer('X', 'RIGHT', 1).next_move(b)
6
# With the larger lookahead, X knows it must pick column 3!
>>> AIPlayer('X', 'LEFT', 2).next_move(b)
3
# The tie-breaking strategy doesn't matter if there's only one best move!
>>> AIPlayer('X', 'RIGHT', 2).next_move(b)
3
>>> AIPlayer('X', 'RANDOM', 2).next_move(b)
3
```

Playing the game with `AIPlayer` objects!

Because our `AIPlayer` class inherits from `Player`, we can use it in conjunction with our `connect_four` function from Part III.

You can play against an `AIPlayer` by doing something like:

```
>>> connect_four(Player('X'), AIPlayer('O', 'RANDOM', 3))
```

Below some examples in which two `AIPlayer` objects play against each other. And because we're using non-random tie-breaking strategies for both players, you should obtain the same results.

```
>>> connect_four(AIPlayer('X', 'LEFT', 0), AIPlayer('O', 'LEFT', 0))
# omitting everything but the final result...

Player X (LEFT, 0) wins in 10 moves.
Congratulations!
|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

>>> connect_four(AIPlayer('X', 'LEFT', 1), AIPlayer('O', 'LEFT', 1))
# omitting everything but the final result...

Player X (LEFT, 1) wins in 8 moves.
Congratulations!
|O|O| | | | | |
|X|X| | | | | |
|O|O| | | | | |
|X|X| | | | | |
|O|O|O| | | | |
```

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

# The player with the larger lookahead doesn't always win!
>>> connect_four(AIPlayer('X', 'LEFT', 3), AIPlayer('O', 'LEFT', 2))
# omitting everything but the final result...

Player O (LEFT, 2) wins in 19 moves.
Congratulations!
|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
```

Testing Your `AIPlayer` Class

You should write a test to verify that:

- Your `AIPlayer` correctly initializes: you can do this by checking the output of the `__repr__` function
- `max_score_column` returns the correct column, for each kind of tie-breaking strategy
- `scores_for` returns the correct scores for a given board. Feel free to use the examples from lecture when constructing the board!

Please let us know if you find any mistakes, inconsistencies, or confusing language in this document or have any concerns about this and any other CS4 document by posting on Piazza or filling out our anonymous feedback form.