Com S 127x
Fall 2016
Assignment 6

Due Date: Friday, December 9, 11:59 pm (midnight)

THERE IS NO "LATE" DEADLINE - all work must be turned in on December 9th

General information

This assignment is to be done on your own. See the Academic Dishonesty policy in the syllabus for details.

If you need help, see your instructor or one of the TAs. Lots of help is also available through the Piazza discussions.

Introduction

The purpose of this assignment is to practice some more with lists, including 2-dimensional lists, and to try out using the object-oriented style of programming in Python. The system to be implemented is a version of the game Connect Four. The game consists of three modules: **`connect_four.py`** is an implementation of a class representing the game state; **`c4util.py`** is a set of utility functions for working with the game grid, and **`c4_text_ui.py`** is a text-based user interface for trying out the game. The UI module is fully implemented and you should not modify it. The other two modules are partially implemented, and the parts for you to do are marked **`#TODO`**.

Connect Four

The game is a variation of tic-tac-toe. Two players take turns placing markers on a two-dimensional grid; we will refer to the markers as 'x' and 'o'. The object of the game is to get a "run" of some specified number of markers in a row, column, or diagonal. Traditionally the length of a winning run is 4, but we will make that configurable (along with the size of the grid). The difference between Connect Four and tic-tac-toe is that the players only get to select the *column* in which to play, not an arbitrary cell. Selecting a cell automatically plays in the bottom-most empty cell in that column (the row with largest index).

As an example, take a look at a screenshot from the text-based UI:

```
0 1 2 3 4 5 6
- - - - - - -
- - - - - - -
- - - - - - -
- - - o - - -
- - o x x - -
- - x o o x -

It is player 0's turn.
Select a column number:
```

If player 0 now selects column 4, the 'x' is placed in the lowest empty position of column 4, as shown:

```
0 1 2 3 4 5 6
- - - - - - -
- - - - - - -
- - - - - - -
- - - o x - -
- - o x x - -
- - x o o x -

It is player 1's turn.
Select a column number:
```

You can also see the Wikipedia page for an illustration,
https://en.wikipedia.org/wiki/Connect_Four

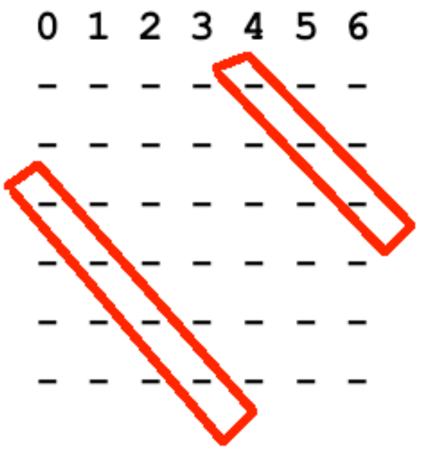
The module c4util.py

As usual, we have a design in which the problem is decomposed into simpler problems that we can code separately as functions and test independently. Much of the work involves working with a 2d grid of characters representing the positions of the markers. As you can see from the example above, one of the tasks is, given a grid and a column number, find the row in which the marker should be placed. This task will be implemented by you as the function **`find_max_empty_row`** in the **`c4util.py`** module. See the function skeleton in that file for details.

You'll also find a function for creating an empty grid and for creating a grid from a list of strings (useful for testing).

Most of the code in **`c4util.py`** is for detecting whether the grid contains a run of a particular character in a row, column, or diagonal. The core of of the strategy is implemented in the function **`check_list(list, ch, size)`**, which returns true if a given list of characters contains a **size**-length run of the given character **ch**. This function is implemented for you. You can see how it can be used in the function **check_cols**: to check whether a column contains a run, just copy the column elements into a list, and call **check_list**. This function, along with **check_rows**, is already implemented for you as an example.

To check for runs along a diagonal is slightly harder. We use the same strategy of first copying the elements into a list, and calling the **check_list** helper function to see whether there is a run of the desired character. The tricky bit is that not all the diagonals are the same length - a diagonal may end at the bottom of the grid or end at the side of the grid, as shown.



As discussed in class before break, the best way to copy diagonal elements into a list is with a while-loop that increments both the row index and column index, beginning with a given starting point. For example, to make a list of right-diagonal elements from a given position at **start_row** and **start_column**, the code would look like:

```
result = []
row = start_row
col = start_col
while row < len(grid) and col < len(grid[0]):
    result.append(grid[row][col])
    row += 1
    col += 1
```

You'll implement this algorithm in the function **make_right_diag_list,** along with the corresponding version for left diagonals. See the **#TODO** markers in **c4util.py**. Then to check *all* diagonals, notice from the illustration above that some diagonals start along the side, and some start along the top. This suggests that you need two loops; for example, for right diagonals, it would be something like this in pseudocode:

```
for each column index c
    check the diagonal starting at row 0, column c
for each row index r
    check the diagonal starting at row r, column 0
```

You can test each of these functions separately. The sample code includes a file `test_util.py` with a couple of examples of simple test cases. Note the use of the function `make_grid_from_strings` to create simple grids for testing.

Finally, notice that the function your game class will actually call to check the grid for a winning run is `check_grid`, which is fully implemented for you but depends on the other functions described above.

The module connect_four.py

The state of the game at any point in time is represented by

- the current grid
- the number of markers in a winning run
- whose turn it is (0 or 1)
- whether the game has been won

In the previous assignment for the hangman game, we used global variables to store the game state. In this design, we will represent the game state as an *object*. In Python, each type of object is defined by a *class*. You can see a skeleton for a class called `ConnectFour` in the `connect_four.py` module. Notice in particular the function `__init__` whose purpose is to initialize the special variables, called *instance variables*, representing the object's state. The `__init__` function is already implemented for you, though you can modify it if you want to make changes in the design. Note also that the simpler methods `whose_turn`, `is_over`, and `get_grid` are already implemented as examples.

The module c4_text_ui.py

Although there is nothing for you to implement in this module, you should read it to see how the game class is used.

If you have questions

For questions, please see the Piazza Q & A pages and click on the folder `hw6`. If you don't find your question answered, then create a new post with your question. Try to state the question or topic clearly in the title of your post, and attach the tag `hw6`. *But remember, do not post any source code for the classes that are to be turned in.* It is fine to post source code for general Python examples that are not being turned in. (In the Piazza editor, use the button labeled "pre" to have code formatted the way you typed it.)

If you have a question that absolutely cannot be asked without showing part of your source code, make the post "private" so that only the instructors and TAs can see it.

Getting started and testing

Before you start this assignment, make sure you understand

- the examples from 11/14 on nested loops, and
- the examples from 11/28 on objects.

See the section on `c4util.py` for ideas about developing and testing testing the utility functions.

The `play()` method in the `connect_four.py` module will depend on the two `c4util` functions `find_max_empty_row` and `check_grid`. However, you can still implement `play()` even if you have errors in your functions for finding diagonal runs - everything should work except that you just won't be able to detect a game that is *won* with a diagonal run.

Documentation and style

Include comments at the top of each file with

- your name, and
- a brief statement of what the program does

For each *function* you write, include a docstring that states what it does. For each global variable that you define, include a comment explaining what it represents.

What to turn in

Turn in the two files `connect_four.py` and `c4util.py`.

Upload your two files to the Homework 6 submission link on Blackboard. Be sure to CHECK your submission history and make sure that you successfully submitted both files. **Remember you can't just upload the files, you have to click the "Submit" button!**