Complica: Connect Four meets awesome

Feb 10, 2018

Complica

Introduction

The objective of this homework is build Complica, a variant of Connect Four that allows for seemingly infinite play. First, let’s get familiar with Connect Four; from Wikipedia:

> *“Connect Four… is a two-player connection game in which the players first choose a color and then take turns dropping colored discs from the top into a seven-column, six-row vertically suspended grid. The pieces fall straight down, occupying the next available space within the column. The objective of the game is to connect four of one’s own discs of the same color next to each other vertically, horizontally, or diagonally.”*

Because Connect Four is played on a vertically positioned physical grid, once a column is full, that column is no longer in play (as there is no more room in the column for player pieces). Complica is an extension of Connect Four that removes that limitation, allowing columns to still be in play even after they are full.

Specifically, once a Complica column is full, subsequent chips into that column push lower level chips out; columns are essentially an infinite queue with limited space. Consider a three row column containing the pieces ‘red’, ‘blue’, and ‘red’, in that order. If a subsequent play by the blue player was to the same column, the contents would be ‘blue’, ‘red’, ‘blue’.

This addition to traditional Connect Four makes for interesting situations. First, a tie could arise – a situation in which both players simultaneously have four of their chip in-a-row. Second, there’s the possibility that a players move could cost them the game: imagine a column “slides” down, creating four-in-a-row for the opponent.

The `layout` module

Before getting started it is worth mentioning the `layout` module. This is code that has been provided to make your life easier, and final code more generic (that is, easier to grade). Understanding everything that is going on in the file is not important. How to use what is going on in the file is.

The first thing your implementation should do is `import layout`. All variables will be accessible from within your implementation by prepending the name with `layout`; for example:

```
import layout
print(layout.rows)
```

The module can be downloaded here. When developing your implementation, you will need to import the file; to do so successfully, layout.py must be in the same directory as your implementation.

Rather than listing the different variables available in this module, it is probably easiest to understand those variables in the context of the implementation.

Implementation (70 points)

The implementation has been broken down into parts. Each part has a point value so that if you are unable to complete the entire assignment, you know the (maximum amount of) partial credit to expect. While it is possible to perform these tasks out of order and complete the game, you might find doing so to be a bit harder.

Making the board (10 / 70)

You should start by making the board. The board should be represented using a list containing (sub)lists of strings. By default, the game is played on a 6-by-7 grid, however your game should be size agnostic. To help with this, use the `rows` and `columns` variables in `layout` when creating the board. To get an idea what those values are, you can print them:

```
import layout
print(layout.rows, layout.columns)
```

Initially, the contents of the sub-lists should be a spaces. Again, there is a variable in `layout` to help with this: `layout.marker.open`. Setting the contents of the sub-lists to this value, rather than the literal string ‘ ‘ is the suggested approach.

Printing the board (10 / 70)

Next, develop code that prints the board using decorations. Each cell should be surrounded by characters such that it appears to be a box. For example, an empty 3-by-3 board would be printed as follows:

```
+-+-+-+
| | | |
+-+-+-+
| | | |
+-+-+-+
| | | |
+-+-+-+
```

Again, the `layout` module can help with the decorators:

Variable	Value
`layout.board.side`	`|`
`layout.board.top`	`-`
`layout.board.corner`	`+`

For example, rather than

```
print('+-+')
```

you can use

```
print(layout.board.corner + layout.board.top + layout.board.corner)
```

While this is more code, it has the advantage that if you want to change the way the board looks, there is only one place – the value of the variables in `layout` – that needs to be changed.

Gameplay (30 / 70)

Complica is a turn-based game. There are two players and each player gets a chance to perform an action. In this case, that action is choosing a column in which to play their piece. One player will be interactive: the column decision in which to “drop” the piece should come from user input. You can assume that the players input will be within range; that is, a value between zero and one less-than the number of columns. The second player will be the computer, who makes its next-play decision randomly.

When a column decision has been made, the piece should take up the next available slot within the specified row:

```
+-+-+-+-+-+-+
| | | | | | |
+-+-+-+-+-+-+
| | | | | | |
+-+-+-+-+-+-+
| | | | | | |
+-+-+-+-+-+-+
Player 1, choose your column: 1
+-+-+-+-+-+-+
| | | | | | |
+-+-+-+-+-+-+
| | | | | | |
+-+-+-+-+-+-+
| |x| | | | |
+-+-+-+-+-+-+
```

If a player chooses a position in which there are no available slots, existing pieces “beneath” the piece should be moved down, and the final, bottom, piece removed completely:

```
+-+-+-+-+-+-+
| |o| | | | |
+-+-+-+-+-+-+
| |x|o| |x| |
+-+-+-+-+-+-+
| |o|o|x|o| |
+-+-+-+-+-+-+
Player 1, choose your column: 1
+-+-+-+-+-+-+
| |x| | | | |
+-+-+-+-+-+-+
| |o|o| |x| |
+-+-+-+-+-+-+
| |x|o|x|o| |
+-+-+-+-+-+-+
```

After a player selects their column, the piece should be placed and the board should be printed. When the computer selects a column, that column choice should be announced prior to printing the board. Again, the layout module contains variables that you can use for the player markers:

Variable	Value
`layout.marker.player_1`	`x`
`layout.marker.player_2`	`o`

At this point, you should have an interactive, back-and-forth, with the computer. The next task is to decide the winner, but for now you should be able to continue laying pieces, and having the computer lay pieces, forever. Think about the repetition constructs that allow us to do such a thing. Further, think about how to structure your code such that those constructs allow for uninterrupted interaction.

Determining a winner (20 / 70)

As is the case with Connect Four, to win in Complica a player must have four consecutive pieces in any position on the board. For this version of the game, however, your implementation should be in-a-row agnostic. That is, the number of consecutive pieces that determine a winner should be based on the value of `layout.in_a_row`. By default, the value of `layout.in_a_row` is four. You should not assume that we will use this value when grading.

Direction check	Marks
Horizontal	5
Vertical	5
Left-diagonal	5
Right-diagonal	5

Once a winner has been determined, a message announcing the victory should be made and the game should stop.

Expectations (30 points)

1. The code that you submit should run. Even if you have not implemented all parts of the assignment, the subset of parts that are implemented should be free of Python errors.

2. The `layout` module must be used when generating the board, when placing chips, and when calculating the winner. These values will be altered when testing, and your code should work regardless.

3. There should be comments at the top of the program specifying your name, and the purpose of the program. Additionally, there should be comments throughout the code announcing which sub-problem this section of code is implementing. If you use a concept that has not been covered in class, something you found online perhaps, you should add a comment explaining what is going on. Without an explanation, we can only assume you have cheated.

Suggestions

For testing purposes, it is okay if you alter the layout file a bit. Changes you might want to consider:

- Altering the values of `rows` or of `columns` to ensure your board is created and printed correctly;

- Changing the value of `layout.in_a_row` to check that your winner determination is accurate.

It might also be a good idea to not make the computer random at first. This way you can test certain board configurations easily. Once you are confident that things are working, make the computer choice random to test full game-play.

Finally, keep in mind that because pieces may shift, it is possible for one players move to render the opposing player victorious!

Submission

A single Python file should be submitted via NYU Classes, against the corresponding homework assignment. There is no need to submit a `layout` file.

jerome.white@nyu.edu