Connect Four with a Computer Opponent

Overview

The aim of this coursework is to code a Python 3 module that implements the famous Connect Four game. Here is a short description of the game, taken from Wikipedia in slightly modified form:

Connect Four is a game in which two players take turns dropping (coloured) 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 be the first to form a horizontal, vertical, or diagonal line of four of one's own discs.
See https://en.wikipedia.org/wiki/Connect_Four for more details.

The Python module should allow the game to be played either by two humans, a human against a computer, or by the computer against itself. The module should also allow the current game to be saved, and a previously saved game to be loaded and continued.

In this project you are required to use the data structures and functions specified in the tasks below. These tasks are ordered logically and in (approximately) increasing level of difficulty; it is highly recommended to follow these tasks in that order.

Frequently asked questions

What can/cannot be used for this coursework?

The Python knowledge contained in the course notes is sufficient to complete this project. While you are allowed to consult the web for getting ideas about how to solve a specific problem, the most straightforward solution may already be in the course notes. It is not allowed to copy-and-paste several lines of computer code from an online source without clearly indicating their origin. If you really need to use an online source, you must indicate this for example like so:

```
# the following 3 lines follow a similar code on
# http://webaddress.org/somecode.htm as retrieved on 24/04/2017
```

Let's be clear about **what is not allowed:**

You are NOT allowed to send/give/receive Python code to/from classmates and others.
This project is the equivalent of a standard exam and it counts 70% towards your final mark.
Consequently, the standard examination rules apply to this project.

Once your Python module is submitted via the Blackboard system, it will undergo plagiarism tests:

1.) The turn-it-in system automatically checks for similarities in the codes among all students (without syntactical comparisons).
2.) Your lecturer compares the syntax of all submitted codes (interestingly, with the help of another Python program). Hence, **refrain from communicating any code with others.**

Note that even if you are the originator of the work (and not the one who copied), the University Guidelines require that you will be equally responsible for this case of academic malpractice and may lose all marks on the coursework (or even be assigned 0 marks for the overall course).

How is this coursework assessed?

There are several factors that go into the assessment:

- First of all, it will be checked whether you have followed the tasks and format specified below, using the prescribed function names, variable names, and data structures.
- Second, your module will be tested manually by performing a number of predefined game moves, loading/saving a couple of predefined boards, and testing the computer opponent.
- Further, all module functions will be tested automatically by another Python program (so-called unit testing). This is why it is important that your module "does not do anything" when it is imported into another program, and that you strictly follow the format of the functions specified in the tasks below (otherwise some of the tests may fail and you may lose marks).
- It is also checked whether each function/line of code is free of bugs and the program runs without crashing. Functionality will be one of the main factors in the assessment.
- It will be tested if your code reacts to exceptional user inputs in an appropriate manner (using Exceptions). It should be impossible to crash the code.
- It will be checked if your module is properly documented, and if the main program and all functions have meaningful docstrings. In particular, each function must explain its own inputs and returned values so that there is no room for misinterpretation.
- Further marks will be given on code efficiency, readability, and strategy.

When and how to submit the coursework?

The complete coursework can be completed and submitted as **a single Python file named `"connect4_{yourname}.py"`.** Make sure to replace `{yourname}` with your actual name! The submission will be via Blackboard and **the strict deadline is Thursday, May 11th, at 1pm.** You can resubmit your coursework as often as you like, but only the last submission counts.

Below are the Tasks 0-10 of the coursework

Task 0. Prepare the module and understand the `game` dictionary

The tasks in this project description specify the various functions to be implemented in your module. The main function of your module is called `play()` and it will initiate and control the game flow (i.e., starting by asking for the users' names and then, in turns, performing the users' moves). We do not need to worry about this `play()` function until later in Task 8. For now, simply prepare your Python project by copying the below code into a fresh "`connect4_{yourname}.py`" file.

```
"""
Add a description of the module.
Also mention your name and student id.
"""

from copy import deepcopy # you may use this for copying a board

def newGame(player1,player2):
    """
    Make sure you write meaningful doctrings!
    """
    game = {}
    # TODO: Initialize dictionary for a new game
    return game

# TODO: All the other functions of Tasks 2-10 go here.
# USE EXACTLY THE PROVIDED FUNCTION NAMES AND VARIABLES!

# ------------------- Main function --------------------
def play():
    """
    TODO in Task 8. Make sure to write meaningful docstrings!
    """
    print("*"*55)
    print("***"+" "*8+"WELCOME TO STEFAN'S CONNECT FOUR!"+" "*8+"***")
    print("*"*55,"\n")
    print("Enter the players' names, or type 'C' or 'L'.\n")
    # TODO: Game flow control starts here

# the following allows your module to be run as a program
if __name__ == '__main__' or __name__ == 'builtins':
    play()
```

Your whole coursework project can be completed using this single file. You "only" need to replace the `TODO` comments with the actual code. Make sure that all code is contained in functions so your module "does not do anything" when it is imported into another Python program.

So how should we "make the Connect Four game play?"

We first have to clarify what constitutes a "state" of this game, and then we need to implement the transitions of one state into another. Note that at any stage of playing, the current game situation is completely described by the factors:

- Who are the two players (i.e., a player's name or if it is a computer player)?
- Which positions on the 7-by-6 (columns-by-rows) board are occupied by which player?
- Whose turn is next?

In order to store this information, we will use a dictionary called `game`. Here is an example:

```
game = {
    'player1' : 'Stefan',
    'player2' : 'C',
    'who' : 1,
    'board' : [[0,0,0,0,2,0,0],
               [0,0,0,0,1,0,0],
               [0,0,0,0,1,0,0],
               [0,0,0,0,2,0,0],
               [0,0,0,2,1,0,0],
               [0,0,2,1,2,1,0]]
}
```

The `game` dictionary has precisely four key-value pairs as follows:

- `player1`: a nonempty string representing the name of player 1 (in this example `'Stefan'`); if the value is the single letter `'C'` then player 1 is controlled by the computer
- `player2`: a nonempty string representing the name of player 2; if the value is the single letter `'C'` (as in this example) then player 2 is controlled by the computer
- `who`: an integer which is either 1 or 2 representing whose turn is next; so in this example it's player 1's (Stefan) turn to make the next move
- `board`: a list of six elements, each of which is a list with seven integer entries (i.e., a two-dimensional array). The first list `board[0]` represents the first upper row of the board, with the entry `board[0][0]` referring to the upper left position, `board[0][1]` to the second position in the first row, and so on. Entry `board[5][6]` corresponds to the position in the bottom right of the board. The integer entries are either 0, 1, or 2, with 0 representing an empty position, and 1 or 2 representing a position occupied by player 1 or 2, respectively.

Task 1. Initialize a new game: `newGame(player1,player2)`

Write a function `newGame(player1,player2)` which takes two string parameters `player1` and `player2` corresponding to the two players' names. The function returns a `game` dictionary as specified in Task 0. In this dictionary all the positions of the board are empty (i.e., all are set to the integer 0), the players' names are set to the input parameters of the function, and the variable `who` is set to the integer 1 (in a new game, player 1 will always make the first move).

Task 2. Print a nicely formatted board: `printBoard(board)`

Write a function `printBoard(board)` which takes a list of lists as an input and prints the 7x6 Connect Four board in a nicely formatted fashion. The input parameter `board` is of the same format as the corresponding value in the `game` dictionary specified in Task 0. Board positions which are not occupied by any player should be printed empty. Positions occupied by player 1 should be marked by an "X", and positions occupied by player 2 should be marked by an "O". For better orientation, the function should also print a numeration of the columns 1,2,...,7 above the board.

Example. When called with the `board` defined within the example `game` dictionary from Task 0, the function should print something like this:

```
|1|2|3|4|5|6|7|
+-+-+-+-+-+-+-+
| | | | |O| | |
| | | | |X| | |
| | | | |X| | |
| | | | |O| | |
| | | |O|X| | |
| | |O|X|O|X| |
```

Note. Internally we will work with the standard Python indexing and enumerate the board's columns and rows starting from 0. To the user the columns will only be *displayed* with numbers starting from 1.

Task 3. Load a game state: `loadGame()`

Write a function `loadGame()` that takes no input arguments. The function attempts to open the text file `'game.txt'` and returns its content in form of a `game` dictionary as specified in Task 0. The format of `'game.txt'` is as follows:

- Line 1 contains a string corresponding to `player1` (either a name or the letter `'C'`)
- Line 2 contains a string corresponding to `player2` (either a name or the letter `'C'`)
- Line 3 contains an integer 1 or 2 corresponding to the value of `who`
- Lines 4-9 correspond to the six rows of the Connect Four board, starting with the upper row. Each line is a comma-separated string of 7 characters, each character being either 0, 1, or 2.

The function should raise an Exception if the file `'game.txt'` cannot be loaded or its content is not of the correct format.

Example. The `game` dictionary from Task 0 would result from loading the following `'game.txt'` file:

```
Stefan
C
1
0,0,0,0,2,0,0
0,0,0,0,1,0,0
0,0,0,0,1,0,0
0,0,0,0,2,0,0
0,0,0,2,1,0,0
0,0,2,1,2,1,0
```

Note. The implementation of this `loadGame()` function is such an relatively early task in this project because, once it's working, you can use it for easily testing your code in the following tasks. In order to create a certain game situation, like for example "No valid moves left", you don't need to actually play the game until this situation occurs. Just make your own `'game.txt'` file to create and test any game situation you like!

Task 4. Get a list of all valid moves: `getValidMoves(board)`

Write a function `getValidMoves(board)` which takes a list of lists as an input. This input list represents the Connect Four board and is of the same format as the corresponding value in the `game` dictionary specified in Task 0.

The function returns a list of integers between 0,1,...,6 corresponding to the indices of the board columns which are not completely filled. Every integer should appear at most once. If no valid move is possible (i.e., the board is completely filled), the function returns an empty list.

Example. When called with our example board from above, the function should return the list `[0, 1, 2, 3, 5, 6]`, or a permutation thereof. Note that the column with index 4 is excluded as this column is already completely filled in this example.

Task 5. Make a move: `makeMove(board,move,who)`

The function takes as input a list `board` representing the Connect Four board, an integer `move` between 0,1,...,6, and an integer `who` with possible values 1 or 2. The parameter `move` corresponds to the column index into which the player with number `who` will insert their “disc”. The function then returns the updated `board` variable.

The function does not need to perform any checks as to whether the `move` input is valid. You can assume that it is only called with valid moves.

Example. When `makeMove(board,3,1)` is called with the example board from above, the returned board (as outputted by `printBoard`) should be:

```
|1|2|3|4|5|6|7|
+-+-+-+-+-+-+-+
| | | | |O| | |
| | | | |X| | |
| | | | |X| | |
| | | |X|O| | |
| | | |O|X| | |
| | |O|X|O|X| |
```

Task 6. Check for a winner: `hasWon(board,who)`

The function takes as input a list `board` representing the Connect Four board and an integer `who` with possible values 1 or 2. The function returns `True` or `False`. It returns `True` if the player with number `who` occupies four adjacent positions which form a horizontal, vertical, or diagonal line. The function returns `False` otherwise.

Example. When called with the following board configuration, `hasWon(board,1)` should return `True`, whereas `hasWon(board,2)` returns `False`.

```
|1|2|3|4|5|6|7|
+-+-+-+-+-+-+-+
| | | | |O| | |
| | | | |X| | |
| | |X|X|X| | |
| | |O|X|O| | |
| | |X|O|X|O| |
| | |O|X|O|X|O|
```

Task 7. An easy computer opponent: `suggestMove1(board,who)`

Write a function `suggestMove1(board,who)` which takes as inputs a list `board` representing the Connect Four board and an integer `who` with possible values 1 or 2. The function returns an integer between 0,1,...,6 corresponding to a column index of the board into which player number `who` should insert their "disc". This column index is determined as follows:

- First check if among all valid moves of player number `who` there is a move which leads to an immediate win of this player. In this case, return such a winning move.
- If there is no winning move for player number who, we will try to prevent the other player from winning. This is done by checking if there is a winning move for the other player and returning it.
- Otherwise, if there is no immediate winning move for both players, the function simply returns a valid move (for example, the first one in the list of valid moves).

Notes.

- The function `suggestMove` does not need to check whether there is a valid move left. You can assume it is only called with a board that is not completely filled.
- For this task you should be able to use the `getValidMoves`, `makeMove`, and `hasWon` functions written earlier. One approach is to first get the list of valid moves and then try to make every single one of them, followed by a check if a move leads to a win of the respective player.
- You should not change the original `board` variable when testing for possible moves. Therefore you may want to create a deep copy of the board as follows:

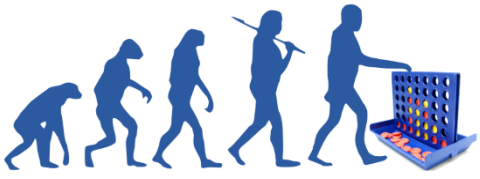
  ```
  board2 = deepcopy(board)
  ```

- Finally, it is advisable to try making this function as efficient as possible. A human player will not want to wait for more than one or two seconds to get the computer's move. Try to call `getValidMoves` at most once and also return from the function as early as possible.

Example. When called with the following board configuration, `suggestMove1(board,1)` should return either 2 or 6 (the column indices which lead to an immediate win of player 1). The call `suggestMove1(board,2)` should also return 2 or 6, because there is no immediate winning move for player 2 and hence we will *try* to prevent player 1 from winning.

```
|1|2|3|4|5|6|7|
+-+-+-+-+-+-+-+
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
|O| | | | | | |
|O|O| |X|X|X| |
```

Note. This strategy is clearly not very good, at best at the playing level of a small child! Player 2 should have prevented Player 1 to form the row of three `X|X|X` in the first place, and now the game can easily be won by Player 1. However, we don't need to worry about this strategy now as we can always refine it later (and will actually do so in the last task). For now we have everything in place to make the game run!



Task 8. Play a game: `play()`

If all your functions from the previous tasks are working as expected, you are now ready to write the main part of your module. This function will take care of the overall operation of the game and needs to do the following things:

1) When `play()` called, it will first print a welcome message to the user.
2) It will then ask for the names of player 1 and player 2. These names are inputted as nonempty strings and their first letter should be automatically capitalized. If the user enters an empty string, the program will keep on asking for the user name.
3) If one of the players' names (or both) is the letter `'C'`, this will mean in the following that the corresponding user is played automatically by the computer.
4) If the name of player 1 is entered as the letter `'L'` (for load), the program will skip asking for the name of player 2 and attempt to load a game dictionary from `'game.txt'` and go to Step 6. If the loading fails the game just ends with an error message.
5) The code creates a new game dictionary with the two players' names, an empty board, and with player 1 being active. With the `game` structure being set, the game play can begin, with both players taking turns alternatingly.
6) If the active player is human, the program asks which column they want to select. The column is specified by the user as an integer 1,2,…,7. The program needs to check that the user input corresponds to a valid move and otherwise print a warning message and repeat asking for a valid input.
7) If the active player's name is `'C'`, the program will make a move automatically.
8) If after a move the program finds that a player has won, the game prints this information and then ends.
9) If there is no valid move left, the game prints that there was a draw and ends.
10) Otherwise, the active player switches and the program continues with Step 6.

Note. At the end of this document, page2 9-10, there is an example output of the whole program. Note that if both `player1` and `player2` have the value `'C'`, the computer will play against itself until one player wins or a draw occurs.

Task 9. Save a game: `saveGame(game)`

Write a function `saveGame(game)` that takes as input a `game` dictionary as specified in Task 0, and writes its content into the text file `'game.txt'` in the format specified in Task 3. If saving fails (for whatever reason), the function will return an exception.

Once you have the `saveGame` function working, go back to your `play()` function and modify it so that when the user is asked for entering a move 1,2,…,7 in Step 6, the program also accepts the letter `'s'` as a command for saving the current game. If saving fails within the `play()` function, the program will just print a warning and continue with the game.

Task 10. A better computer opponent: `suggestMove2(board,who)`

This is the last (and perhaps most exciting) task of this project. Write a function `suggestMove2(board,who)` which takes as inputs a list `board` representing the Connect Four board and an integer `who` with possible values 1 or 2. It returns an integer between 0,1,...,6 corresponding to a column index of the board into which player number `who` should insert their "disc".

You are completely free to come up with your own playing strategy. The only two requirements are:

1) Your own strategy should hopefully be better than the easy one implemented in `suggestMove1` before.
2) This `suggestMove2` function **should not take longer than about 5 seconds** (measured on https://repl.it/languages/python3) to return with a suggested move.

Note. Connect Four on the 7x6 board is a so-called "solved" game which means that if both players play perfectly, then player 1 is guaranteed to win if he/she places the first chip in the centre position of the board. You can read more about this and find references on the Wikipedia page https://en.wikipedia.org/wiki/Connect_Four. It is by no means expected that your own strategy makes the computer play perfectly, but it should be better than `suggestMove1`.

End of coursework description

Example output

```
**********************************************************
***         WELCOME TO STEFAN'S CONNECT FOUR!           ***
**********************************************************

Enter the players' names, or type 'C' or 'L'.

Name of player 1: Stefan
Name of player 2: c
Okay, let's play!

|1|2|3|4|5|6|7|
+-+-+-+-+-+-+-+
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |

Stefan (X): Which column to select? 4

|1|2|3|4|5|6|7|
+-+-+-+-+-+-+-+
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | |_| | | |
| | | |X| | | |

Computer (O) is thinking... and selected column 1.

|1|2|3|4|5|6|7|
+-+-+-+-+-+-+-+
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | |_| | | |
|O| | |X| | | |

Stefan (X): Which column to select? 9
Invalid input. Try again!
Stefan (X): Which column to select? 5
```

```
|1|2|3|4|5|6|7|
+-+-+-+-+-+-+-+
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
|O| | |X|X| | |

Computer (O) is thinking... and selected column 3.

|1|2|3|4|5|6|7|
+-+-+-+-+-+-+-+
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
|O| |O|X|X| | |

Stefan (X): Which column to select? s
The game has been saved to a file.

Stefan (X): Which column to select? 2

|1|2|3|4|5|6|7|
+-+-+-+-+-+-+-+
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
|O|X|O|X|X| | |

Computer (O) is thinking... and selected column 6.

|1|2|3|4|5|6|7|
+-+-+-+-+-+-+-+
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
|O|X|O|X|X|O| |


[. . . A COUPLE OF MOVES LATER . . .]


Stefan (X): Which column to select? 2

|1|2|3|4|5|6|7|
+-+-+-+-+-+-+-+
| | | | | | | |
| | | | | | | |
| | | | |O| | |
| |X|O|O|X| | |
| |X|O|X|X| | |
|O|X|O|X|X|O| |

Computer (O) is thinking... and selected column 3.

|1|2|3|4|5|6|7|
+-+-+-+-+-+-+-+
| | | | | | | |
| | | | | | | |
| | |O| |O| | |
| |X|O|O|X| | |
| |X|O|X|X| | |
|O|X|O|X|X|O| |

Computer (O) has won!
```