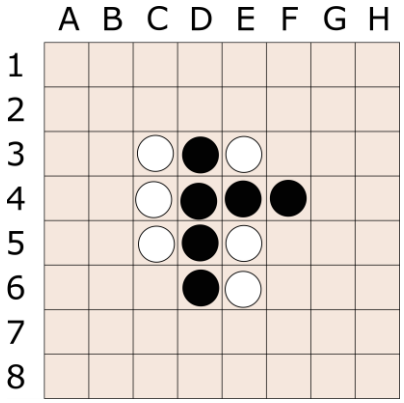
Gomoku with a Computer Opponent

Overview

The aim of this coursework is to code a Python 3 module that implements the famous Gomoku game. This project is worth 70% of the final mark for *MATH20621 Programming with Python*.

Gomoku (also called *Five in a Row*) is a game in which two players take turns marking the spaces on a regular grid (usually using black and white pieces). There are different versions of this game with different grid sizes. In this project, our Gomoku will be played on a 8-by-8 grid. The objective of the game is to be the first to form a horizontal, vertical, or diagonal line of five of one's own pieces.
See https://en.wikipedia.org/wiki/Gomoku



The Python module (= a collection of functions) 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 for a game to be loaded from a file 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 recommended to follow the tasks in that order.

Frequently asked questions

What can and 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 like so:

```
# the following 3 lines follow a similar code on
# http://webaddress.org/somecode.htm as retrieved on 27/11/2022
```

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 Turnitin system automatically checks for similarities in the codes among all students (without syntactical comparisons).
2.) Your lecturer compares the syntax of all submitted codes using specialized software able to detect several forms of plagiarism. Hence, **refrain from communicating any code with others.**

WARNING: 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 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?

- It will be checked whether you have followed the tasks and format specified below, using the prescribed function names, variable names, and data structures.
- 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 will be 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, making use of Exceptions whenever indicated. 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 fully explain its own inputs, its return values, and possible exceptions. There should be no room for misinterpretation. Also check that `print(functionname.__doc__)` returns the docstring.
- Further marks will be given on code efficiency.

The rough split (subject to scaling) between the total marks is 65% for unit testing and manual inspection and 35% for documentation and code efficiency.

When and how to submit the coursework?

The complete coursework can be completed and submitted as **a single Python file named `"gomoku_yourname.py"`.** Replace `yourname` with your actual name. The submission will be via Blackboard and **the strict deadline is Friday, December 16th, at 1pm.** You can resubmit your coursework as often as you like, but only the last submission counts.

Below are the Tasks 0-11 of the coursework

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

The tasks in this project 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 turn, performing the users' moves). We do not need to worry about this `play()` function until later in Task 10. For now, simply prepare your Python project by copying the below code into a fresh "`gomoku_yourname.py`" file.

```
"""
A Python module for the Gomoku game.
TODO: Add a description of the module.

Full name: Peter Pan
StudentId: 123456
Email: peter.pan@student.manchester.ac.uk
"""

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

def newGame(player1, player2):
    """
    TODO in Task 1. Write meaningful docstring!
    """
    game = {}
    # TODO: Initialize dictionary for a new game
    return game

# TODO: All the other functions of Tasks 2-11 go here.
# USE EXACTLY THE PROVIDED FUNCTION NAMES AND VARIABLES!

# ------------------- Main function --------------------
def play():
    """
    TODO in Task 10. Write meaningful docstring!
    """
    print("*"*55)
    print("***"+" "*8+"WELCOME TO STEFAN'S GOMOKU!"+" "*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 just 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 can we "make the Gomoku 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 8-by-8 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' : [[0,0,0,0,0,0,0,0],
               [0,0,0,0,0,0,0,0],
               [0,0,1,2,1,0,0,0],
               [0,0,1,2,2,2,0,0],
               [0,0,1,2,1,0,0,0],
               [0,0,0,2,1,0,0,0],
               [0,0,0,0,0,0,0,0],
               [0,0,0,0,0,0,0,0]]
}
```

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 capital 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 capital 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 eight elements, each of which is a list with eight 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[7][7]` 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 nonempty string parameters `player1` and `player2` corresponding to the players' names. The function returns a `game` dictionary of the format specified in Task 0. In this dictionary all the positions of the board are empty (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 `board` as argument and prints the 8-by-8 Gomoku board in a nicely formatted fashion. The list `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 with an "X", and positions occupied by player 2 should be marked with an "O". For better orientation, the function should also print a numeration of the columns `a,b,…,h` and rows `1,2,…,8`. The function does *not* need to perform any checks whether the argument `board` is of the correct form.

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

```
  |a|b|c|d|e|f|g|h|
  +-+-+-+-+-+-+-+-+
1|  |  |  |  |  |  |  |
2|  |  |  |  |  |  |  |
3|  |  |X|O|X|  |  |  |
4|  |  |X|O|O|O|  |  |
5|  |  |X|O|X|  |  |  |
6|  |  |  |O|X|  |  |  |
7|  |  |  |  |  |  |  |  |
8|  |  |  |  |  |  |  |  |
  +-+-+-+-+-+-+-+-+
```

Note: Internally our code will always work with the standard Python indexing and enumerate the board's rows and columns starting from 0. To the user the rows/columns will only be *displayed* with numbers/letters starting from 1/a. In order to simplify the conversion between both of these representations for later tasks, the following two tasks will provide some helper functions.

Task 3. Convert position string to indices: `posToIndex(s)`

Write a function `posToIndex(s)` that takes a string `s` as argument and returns a tuple `(r,c)` with `r` and `c` corresponding to the indices of the associated board row and column position. The string `s` should contain a single-digit integer from `1,…,8` and a single letter from `a,…,h` (or `A,…,H`), possibly with empty spaces surrounding them. If the provided string cannot be converted, the function should raise a `ValueError` exception.

Examples: `posToIndex('C4')`, `posToIndex('4c')`, `posToIndex('  c 4  ')` should all return the tuple (3,2). But, `posToIndex('  x  9')` or `posToIndex('c3  3')` should raise a `ValueError`.

Task 4. Convert indices to position string: `indexToPos(t)`

Write a function `indexToPos(t)` that takes as argument a tuple `t` of the form `(r,c)` with `r` and `c` corresponding to the indices of the associated board row/column position (each an integer in `0,…,7`). The function returns a 2-character string corresponding to the board column/row using a single letter from `a,…,h` and a single-digit integer from `1,…,8`. The function does *not* need to perform a check whether the argument `t` is of a valid format.

Example: `indexToPos((3,2))` should return the string `'c4'`.

Task 5. Load a game from a file: `loadGame(filename)`

Write a function `loadGame()` that takes a single string input argument. The function attempts to open the text file of the name `filename` and returns its content in form of a `game` dictionary as specified in Task 0. You can assume that the file to be loaded is in the same folder as your .py script and there is no need to specify any folders. The format of a valid text file is as follows:

- Line 1 contains a nonempty string corresponding to `player1` (either a name or the letter `'C'`)
- Line 2 contains a nonempty 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 4-11 correspond to the eight rows of the Gomoku board, starting with the upper row. Each line is a comma-separated string of 8 characters, each character being either 0, 1, or 2.

The function should raise a `FileNotFoundError` exception if the file cannot be loaded. If the file's content is not of the correct format, a `ValueError` exception should be raised.

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

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

Note: This `loadGame()` function is such a 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 and load it to create any situation you like!

Task 6. 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 Gomoku 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 tuples of the form `(r,c)` with `r` and `c` corresponding to the indices of the associated board row/column position (each an integer in `0,…,7`). Every possible move 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 (see Task 2), the function should return a list with 52 tuples.

Task 7. Make a move: `makeMove(board,move,who)`

The function takes as input a list `board` representing the Gomoku board, a tuple `move` of the form `(r,c)`, and an integer `who` with possible values 1 or 2. The tuple `move` contains the row and column indices onto which the player with number `who` will place their "piece". 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,(1,3),2)` is called with the example board from above, the returned board (as outputted by `printBoard`) should be:

```
 |a|b|c|d|e|f|g|h|
 +-+-+-+-+-+-+-+-+
1|  |  |  |  |  |  |  |
2|  |  | O|  |  |  |  |
3|  |  |X|O|X|  |  |  |
4|  |  |X|O|O|O|  |  |
5|  |  |X|O|X|  |  |  |
6|  |  |  |O|X|  |  |  |
7|  |  |  |  |  |  |  |  |
8|  |  |  |  |  |  |  |  |
 +-+-+-+-+-+-+-+-+
```

Task 8. Check for a winner: `hasWon(board,who)`

Week 7

The function takes as input a list `board` representing the Gomoku 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 five adjacent positions which form a horizontal, vertical, or diagonal line. The function returns `False` otherwise.

Example. When called with the board configuration shown in Task 7, `hasWon(board,1)` should return `False`, whereas `hasWon(board,2)` returns `True`.

Task 9. An easy computer opponent: `suggestMove1(board,who)`

Write a function `suggestMove1(board,who)` which takes as inputs a list `board` representing the Gomoku board and an integer `who` with possible values 1 or 2. The function returns a tuple `(r,c)` with `r` and `c` corresponding to the indices of the associated board row/column position (each an integer in `0,…,7`) onto which player number `who` should place their "piece". The suggested move 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 for 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).

Hints:

- The function `suggestMove1` 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, hence you need to work with copies. You can create a so-called 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 board configuration shown in Task 2, `suggestMove1(board,1)` should return either `(1,3)` or `(6,3)` as these are the positions that would allow player 2 to win in the next move.

Note: This strategy is clearly not very good, at best at the playing level of a small child! Player 1 should have prevented Player 2 from forming the row of four `O`'s in the first place, and now the game will inevitably be won by Player 2. However, we will not worry about this strategy now as we can always refine it later. For now, we have everything in place to make the game run!

Task 10. 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) The code creates a new `game` dictionary with the two players' names, a prepared board, and with player 1 being active. With the `game` structure being set, the game play can begin, with both players taking turns alternatingly.
5) If the active player is human, the program asks which move they want to make. Their input can be any string that `posToIndex(s)` can handle. 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.
6) If the active player's name is `'C'`, the program will make a move automatically.
7) If after a move the program finds that a player has won, the game prints this information and then ends.
8) If there is no valid move left, the game prints that there was a draw and ends.
9) Otherwise, the active player switches and the program continues with Step 5.

Note: If both `player1` and `player2` have the value `'C'`, the computer will play against itself until one player wins or a draw occurs.

Task 10a. Extend the `play()` function

Go back to your `play()` function from Task 10 and modify it so that:

1) At the beginning of the program, when the name of player 1 is entered as the letter `'L'` (capital L for "load"), the program will skip asking for the name of player 2, ask instead for a filename and attempt to load the game dictionary from that file. If no filename is entered, the file `'game.txt'` is loaded. If the loading fails for whatever reason the game just ends with an error message. If loading is successful, the game continues from the loaded state.

Optional task 11. Computer opponent `suggestMove2(board,who)`

This is the last (and perhaps most exciting) task of this project, yet it is completely optional. Write a function `suggestMove2(board,who)` which takes as input parameters a list `board` representing the Gomoku board and an integer `who` with possible values 1 or 2. Similarly to `suggestMove1` it returns a tuple `(r,c)` with `r` and `c` corresponding to the indices of the associated board row/column position (each an integer in `0,…,7`) onto which player number `who` should place their "piece".

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.

End of coursework description