1. Gomoku (五子棋)

Gomoku, also called Five in a Row. It is traditionally played with Go (围棋) pieces (black and white stones) on a Go board. Players take turns to put black and white Go pieces onto the board (**Player 1 always starts with black colour**).

Free-style Gomoku requires **a line of five or more stones for a win**. The stones could be connected in horizontal, vertical or diagonal directions. For example, in the following picture, player 2 using white won, because he managed to form a line of 5 connected white stones in row 3.

To have a better understating of the spacing in the string format, you may refer to the file "`gomoku_example.txt`".

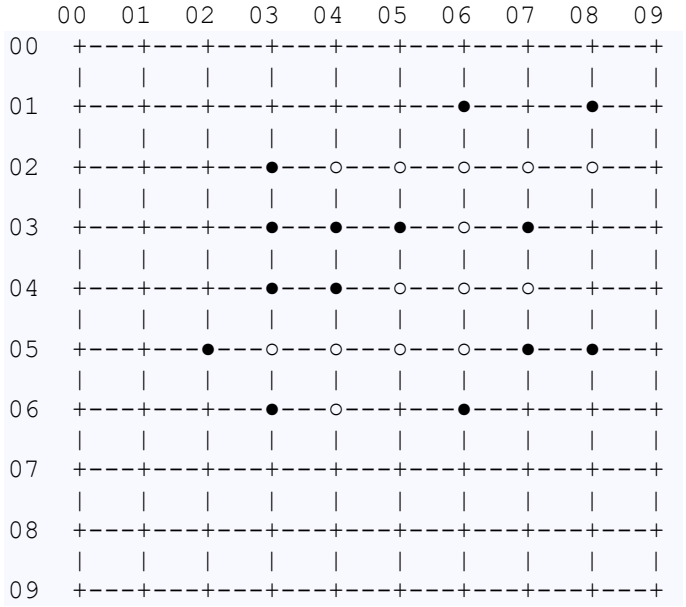
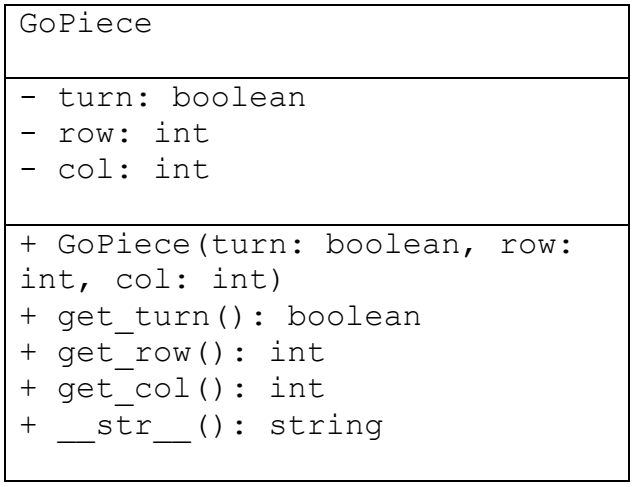


Figure 1 - Gomoku

In the following series of tasks, you are required to implement a python-based Gomoku game using the command-line interface.

Task 1.1 GoPiece Class

Implement the `GoPiece` class based on the following UML class diagram. The descriptions for some class methods can be found below. [4]



Methods	Descriptions

`turn: boolean`	As player take turns to play as black and white, the Boolean value `turn` will determine whose turn it is.
	
	You may consider using the following global dictionaries and functions.
	
	`colour = {True: "black", False: "white"}`
	`colour_symbol = {True: "●", False: "o"}`
	`player = {True: 1, False: 2}`
	
	`def get_player(turn):`
	`  return "Player {} ({})".format(`
	`  player[turn], colour[turn])`

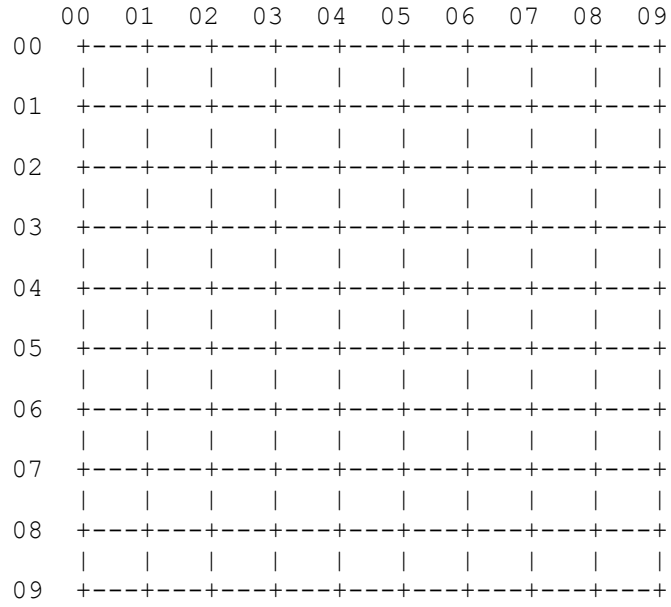
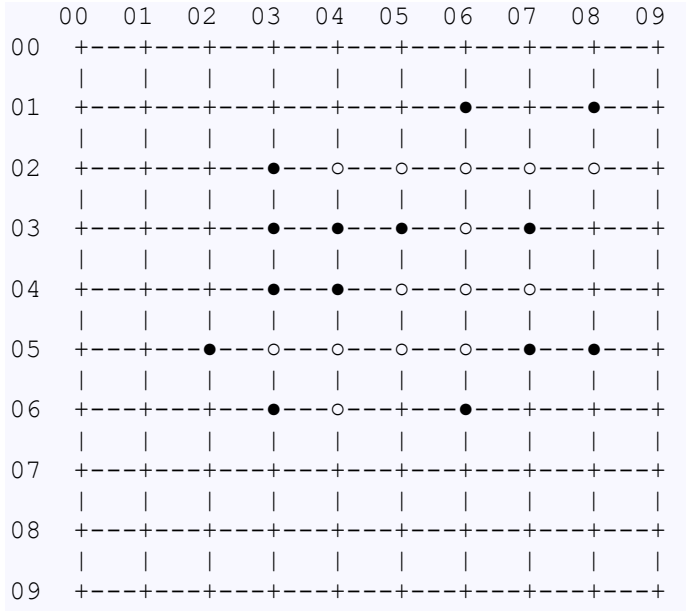
`__str__(): string`	Return a `string` stating the player, colour of the go piece, row and col value. For example:
	
	`"Player 1 (black) plays at (0, 0)."`
	or
	`"Player 2 (white) plays at (0, 1)."`

Task 1.2 Board Class

Implement the `Board` class based on the following UML class diagram. The descriptions for some class attributes and methods can be found below. [15]

Board
- board_size: int - board: list - curr_steps: list - redo_stk: list - turn: boonlean = True - won: boolean = False
+ Board(board_size: int) + reset() + display_board() + is_valid_move(row, col): boolean + new_move(row, col) + check_win(): boolean + undo_move() + redo_move()

Attributes	**Descriptions**
`board_size: int`	Size of the square shape board, if size is 10, then the board dimension should be 10 x 10.
`board: list`	This attribute should be a 2-dimensional `list` initialized with `None` values for every cell. This should be **dynamically** generated based on the `board_size`.
`curr_steps: list`	A `list` implementation of `stack`, keeping track of steps up to the current move. When there is a need to undo a step, the last move should be removed from the top of the `curr_steps`.
`redo_stk: list`	A `list` implementation of `stack`, keeping track of steps have been undone so far. When there is a need to redo a step, the last undone move should be removed from the top of the `redo_stk`, and added to the `curr_steps`. When a new move is performed, the `redo_stk` should be cleared.
`turn: boolean = True`	Boolean value to keep track of player's turn, default set to True, indicating starting with player 1 (black).
`won: Boolean = False`	Boolean value to keep track if the game has already been won by one of the players.
`reset()`	reset the `board`, `curr_steps`, `redo_stack`, `turn` and `won`

Methods	Descriptions
`display_board()`	Print the current `board`. The top row of `col` numbers and left column of `row` numbers should also be dynamically generated based on the `board_size`. When empty, the `board` should look like the following:  *Figure 2 - Empty Board* When the `board` is filled with some Go pieces, it should replace the crossing symbol ("+") with Go piece symbols ("○" or "●"). For example:  *Figure 3 - Change symbol from "+" to "○" or "●"*

`is_valid_move(row, col): boolean`	Check if playing a new piece on (`row`, `col`) position is a valid move.
`new_move(row, col)`	Check if game is not yet finished, and if the move is valid. Perform a new move and add the `go_piece` object into the `board` and `curr_steps` stack. Clear the `redo_stk`. If game is not yet finished, change player turn.
`check_win()`	Based on the last `go_piece` object in the `curr_steps`, check if this move will make the player win the game. If the winning condition is met, return `True`; otherwise return `False`.
`undo_move()`	First check if there is any move to be undone. If there is, undo the move by managing the `curr_steps` stack and the `redo_stk` stack. A success or error message should be printed depending on if the undo operation is performed successfully.
`redo_move()`	First check if there is any move to be redone. If there is, redo the move by managing the `curr_steps` stack and the `redo_stk` stack. A success or error message should be printed depending on if the redo operation is performed successfully.

Task 2. Gomoku Menu (Optional)

In this task you are required to implement the game menu with related operations. You may assume the board's dimension to be 10 rows x 10 columns.

Task 2.1

Write a menu which has the following options. Validation of the user's choice of option is needed. [4]

```
Choose an option below:
1) New Move
2) Undo Move
3) Redo Move
4) Reset
5) Exit
```

Task 2.2

The descriptions for the options can be found below. [7]

Option	Descriptions
`New Move`	At the beginning of a game, Player 1 always starts by using black colour Go pieces. Then the two players will take turns to make new move. Take users' input for the `row` and `col` number. Check if the selected position is available. If it is not available, prompt user to input a new position. If the position is available to be occupied by a new Go piece, add and record this new move, and determine if the winning condition is met. Display how the current `board` looks like.
`Undo Move`	Undo a previous move, if any.
`Redo Move`	Redo a move, if any.
`Reset`	Reset the board and start with player 1 again.
`Exit`	Exit the program.