Connect 4 (ish)

Assignment 1
Semester 1, 2024
CSSE1001

Due date: 28 March 2024, 16:00 GMT+10

1 Introduction

In this assignment, you will implement a text-based version of Connect 4, with some rule modifications inspired by this version developed by Hasbro Inc. The rules of this game are very similar to regular connect four: Two players each have a set of pieces (In our text-based version player 1's pieces are represented by `X`, and player 2's pieces are represented by `O`). Players take turns to place pieces in one of 8 columns. These pieces are affected by gravity and fall into the lowest empty space out of 8 rows within each column. The objective for each player is to be the first to form an unbroken line with 4 of their own pieces. These lines can occur either vertically, horizontally, or diagonally. The twist with this version of the game is that, on their turn, instead of placing a piece at the top of a column, a player may choose to 'pop out' a piece from the *bottom* of a column. All pieces above the removed piece within the chosen column will then 'fall down' one row. This interrupts the opponents plans, and potentially forms an unbroken line of 4 pieces of one kind. A nifty feature of this 'pop out' mechanic is that it also prevents stalemates from occuring. As such, the game is only over when either:

1. One player *wins* by creating an unbroken line (horizontal, vertical, or diagonal) of at least 4 of their own pieces at the end of a turn, while not creating an unbroken line of 4 of the other player's pieces at the end of the same turn.
2. The players *draw* because at the end of a turn *both* players posess an unbroken line (horizontal, vertical, or diagonal) of at least 4 of their own pieces (This can happen when a player pops out a piece).

2 Getting Started

Download `a1.zip` from Blackboard — this archive contains the necessary files to start this assignment. Once extracted, the `a1.zip` archive will provide the following files:

`a1.py` *This is the only file you will submit* and is where you write your code. *Do not* make changes to any other files.

`a1_support.py` *Do not modify or submit this file*, it contains pre-defined constants to use in your assignment. In addition to these, you are encouraged to create your own constants in `a1.py` where possible.

`gameplay/` This folder contains a number of example outputs generated by playing the game using a fully-functional completed solution to this assignment. The purpose of the files in this folder is to help you understand how the game works, and how output should be formatted.

***NOTE:* You are not permitted to add any additional import statements to `a1.py`. Doing so will result in a deduction of up to 100% of your mark.** You must not modify or remove the two import statements already provided to you in `a1.py`. Removing or modifying these existing import statements may result in your code not functioning, and in some cases will result in a deduction of up to 100% of your mark.

3 Gameplay

This section provides an overview of gameplay. Where prompts and outputs are not explicitly mentioned in this section, please see Section 4 and the example games in the `gameplay/` folder provided with this assignment.

The game begins with an empty board of 8 rows separated into 8 columns. Player 1 (`X`) gets to make the first move. Until the end of the game, the following steps occur:

1. The current game board state is displayed.
2. The user is informed whose turn it is to move.
3. The user is prompted to enter a command, and then enters one. See Table 1 for the set of valid commands and the actions performed when they are entered. The `gameplay/` folder provided with this assignment presents specific examples for what to do on each command.
4. If the move is invalid for any reason, the user is shown a message to inform them of why their move was invalid (see Table 2 for all required validity checking and messages for this step), and then the program returns to step 3. If the move is valid, the program progresses to the next step.
5. The board is updated according to the requested move, and the updated board state is displayed.
6. If the game is over (Due to either a win or a draw), the program continues to the next step. Otherwise, the program returns to step 2.
7. When the game is over, the users are informed of the outcome.
8. The users are prompted as to whether they would like to play again. At this prompt, if they enter either 'y' or 'Y', a new game is created (i.e. an empty board is set up and the game returns to player 1's turn) and the program returns to step 1. If they enter anything other than 'y' or 'Y', the program should terminate gracefully (that is, the program should end without causing any errors or exiting the test suite).

Valid Command	**Action to take**
`"{action}{column}"`, where `{action}` may be `"a"`,`"A"`,`"r"` or `"R"` and `{column}` is an integer	If `{action}` is `"a"` or `"A"`, add a piece to the top of the column with number given by `{column}`. If `{action}` is `"r"` or `"R"`, remove a piece from the bottom of the column with number given by `{column}`.
`"h"` or `"H"`	The user is shown a help message, and then prompted for another command
`"q"` or `"Q"`	gameplay does not continue and the program skips to step 8

Table 1: Valid commands and the actions that should be taken. If the command entered by the user does not *exactly* match one of the commands in this table then no action should be taken for step 3 and the program should move directly to step 4

Issue with user input	**Constant in `a1_support.py`**
The move entered does not begin with a valid character, the first character is not followed by a single digit integer, or the command contains any superflous characters.	`INVALID_FORMAT_MESSAGE`
The integer is not a valid column on the board.	`INVALID_COLUMN_MESSAGE`
The move is requesting to add a piece to a full column.	`FULL_COLUMN_MESSAGE`
The move is requesting to remove a piece from an empty column.	`EMPTY_COLUMN_MESSAGE`

Table 2: Constants containing the messages to display when invalid user input is entered. Precedence is **top down** (i.e. if there are multiple issues with user input, only display the message for the one which occurs first in this table).

4 Implementation

Permitted Techniques:

This assesment has been designed to allow you to practice what you have learnt in this course so far. As such, you ***must only use*** the functions, operators and data types presented to you in lectures up to (and including) Topic 4B (Lists). Namely, the following techniques are permitted for use in this assignment:

- Functions (`def`,`return`)
- Basic control structures (`for`, `while`, `if`, `break`)
- Primitive data types (`int`, `str`, `bool` etc.)
- Variable assignment (`=`)
- Arithmetic (`+`,`-`,`*`,\,\\, ,% etc.)
- Comparison (`==`,`<=`,`>=`,`<`,`>`,`!=` etc.)
- Basic Logic (`not`, `and`, `or` etc.)
- `lists` and `tuples`
- `range` and `enumerate`
- `input` and `print`

Using any functions, operators and data types that have not been presented to you in lectures up to (and including) Topic 4B (Lists) will result in a deduction of up to 100% of your mark.

A pinned thread will be maintained on the Ed discussion board with a list of permitted techniques. If you would like clarification on whether you are permitted to use a specific technique, please first check this list. If the technique has not been mentioned, please ask about permission to use the technique in a comment on this pinned thread.

Required Functions

This section outlines the functions you are **required** to implement in your solution (in `a1.py` only). You are awarded marks for the number of tests passed by your functions when they are tested *independently* of one another. Thus an incomplete assignment with *some* working functions may well be awarded more marks than a complete assignment with faulty functions. Your program must operate *exactly* as specified. In particular, your program's output must match *exactly* with the expected output. Your program will be marked automatically so minor differences in output (such as whitespace or casing) *will* cause tests to fail resulting in a *zero mark* for that test.

Each function is accompanied with some examples for usage to help you *start* your own testing. You should also test your functions with other values to ensure they operate according to the descriptions.

The following functions **must** be implemented in `a1.py`. They have been listed in a rough order of increasing difficulty. This does not mean that earlier functions are necessarily worth less marks than later functions. It is *highly recommended* that you do not begin work on a later function until each of the preceding functions can *at least* behave as per the shown examples. You may implement additional functions if you think they will help with your logic or make your code easier to understand.

4.1 `num_hours() -> float`

This function should return the number of hours you estimate you spent (or have spent so far) on the assignment, as a *float*. Ensure this function passes the relevant test on Gradescope *as soon as possible*. The test will only ensure you have created a function with the correct name and number of arguments, which returns a float and does not prompt for input. You will not be marked incorrect for returning the 'wrong' number of hours. The purpose of this function is to enable you to verify that you understand how to submit to Gradescope as soon as possible, and to allow us to gauge difficulty level of this assignment in order to provide the best possible assistance. You will not be marked differently for spending more or less time on the assignment.

If the Gradescope tests have been released, you must ensure this function passes the relevant test before seeking help regarding Gradescope issues for any of the later functions. See Section 5.3 for instructions on how to submit your assignment to Gradescope.

4.2 `generate_initial_board() -> list[str]`

Returns the initial board state (i.e. an empty board state). The board is represented by a list of strings. Each column is represented by a string of characters. The first string in the list represents the leftmost column of the game board, and the last string in the list represents the rightmost column of the game board. The first character of each string represents the top of the associated column, and the last character of the string represents the bottom of the associated column.
Example:

```
>>> generate_initial_board()
['--------', '--------', '--------', '--------', '--------', '--------',
 '--------', '--------']
```

4.3 `is_column_full(column: str) -> bool`

Returns True if the given column is full, and False otherwise. You may assume that `column` will represent a valid column state (i.e. no blank spaces between pieces).
Example:

```
>>> column = "---XOXXX"
>>> is_column_full(column)
False
>>> column = "OXXOOXOO"
>>> is_column_full(column)
True
```

4.4 `is_column_empty(column: str) -> bool`

Returns True if the given column is empty, and False otherwise. You may assume that `column` will represent a valid column state (i.e. no blank spaces between pieces).
Example:

```
>>> column = "--------"
>>> is_column_empty(column)
True
>>> column = "-----XXO"
>>> is_column_empty(column)
False
```

4.5 `display_board(board: list[str]) -> None`

Prints the game board to the terminal with columns separated by pipe characters (—) and numbered below. The printed output must *exactly* match the format as presented in examples. Note that different system fonts may cause spacing to *appear* different on your machine.
A precondition to this function is that the input board will contain exactly 8 strings each with exactly 8 characters. You should not perform any additional validity checking (that is, do not check that the board represents a valid game state).
Example:

```
>>> board = generate_initial_board()
>>> display_board(board)
|-|-|-|-|-|-|-|-|
|-|-|-|-|-|-|-|-|
|-|-|-|-|-|-|-|-|
|-|-|-|-|-|-|-|-|
|-|-|-|-|-|-|-|-|
|-|-|-|-|-|-|-|-|
|-|-|-|-|-|-|-|-|
|-|-|-|-|-|-|-|-|
 1 2 3 4 5 6 7 8
>>> board = ['--------', '----OOOO', 'XXXXXXXX', '--------', '------XO',
 '--------', '---XXOXO', '--------']
>>> display_board(board)
|-|-|X|-|-|-|-|-|
|-|-|X|-|-|-|-|-|
|-|-|X|-|-|-|-|-|
|-|-|X|-|-|-|X|-|
|-|O|X|-|-|-|X|-|
|-|O|X|-|-|-|O|-|
|-|O|X|-|X|-|X|-|
|-|O|X|-|O|-|O|-|
 1 2 3 4 5 6 7 8
>>> board = ['Ashleigh', '        ', '-----W--', 'B----i--', '-r---l--',
 '--a--s--', '---e-o--', '-----n--']
>>> display_board(board)
|A| |-|B|-|-|-|-|
|s| |-|-|r|-|-|-|
|h| |-|-|-|a|-|-|
|l| |-|-|-|-|e|-|
|e| |-|-|-|-|-|-|
|i| |W|i|l|s|o|n|
|g| |-|-|-|-|-|-|
|h| |-|-|-|-|-|-|
 1 2 3 4 5 6 7 8
```

4.6 `check_input(command: str) -> bool`

Returns True if `command` is a well formatted command that is not invalid as described in the first two rows of Table 2, and False otherwise.
Note that user inputs will be 1-indexed (That is, users will enter numbers corresponding to the columns as numbered in the print out by `display_board`). In the event that `command` is ill-formed, this function should also display the relevant error message to the user before returning False. This function should not check whether the command violates any game rules. It is sufficient to assume that the user entered column must be a single digit number.
Example:

```
>>> command = "a1"
>>> check_input(command)
True
>>> command = "r1"
>>> check_input(command)
True
>>> command = "a3"
>>> check_input(command)
True
>>> command = "h"
>>> check_input(command)
True
>>> command = "1r"
>>> check_input(command)
Invalid command. Enter 'h' for valid command format
False
>>> command = "a3 "
>>> check_input(command)
Invalid command. Enter 'h' for valid command format
False
>>> command = "a9"
>>> check_input(command)
Invalid column, please enter a number between 1 and 8 inclusive
False
>>> command = ""
>>> check_input(command)
Invalid command. Enter 'h' for valid command format
False
```

4.7 `get_action() -> str`

This function should repeatedly prompt the user for a command until they enter a command that is valid according to `check_input`, and return the first valid command entered by the user. This function should also result in messages being displayed as described in the specification for `check_input` whenever the user enters an invalid command.
Example:

```
>>> get_action()
Please enter action (h to see valid commands): r-1
Invalid command. Enter 'h' for valid command format
Please enter action (h to see valid commands): a
```

```
Invalid command. Enter 'h' for valid command format
Please enter action (h to see valid commands): r4
'r4'
>>> get_action()
Please enter action (h to see valid commands): g
Invalid command. Enter 'h' for valid command format
Please enter action (h to see valid commands): help
Invalid command. Enter 'h' for valid command format
Please enter action (h to see valid commands): H
'H'
```

4.8 add_piece(board: list[str], piece: str, column_index: int) -> bool

Adds the specified `piece` to the column at the given `column_index` (0-indexed) of the given `board` according to the game rules. The piece will be added to the topmost available space in the requested column. If the requested column is full, then a piece is not added and a message is displayed to the user as described in Table 2. This function should return True if a piece was able to be added to the board, and False otherwise. Note that this function mutates the given board and does *not* return a new board state.
A precondition to this function is that the specified board will contain exactly 8 strings each with exactly 8 characters, and represent a valid game state. Another precondition to this function is that the specified column index will be between 0 and 7 inclusive. The last precondition to this function is that the given piece will be exactly one character in length.
Example:

```
>>> board = ['--------', '----OOOO', 'XXXXXXXX', '--------', '------XO',
'--------', '---XXOXO', '--------']
>>> add_piece(board, "X", 1)
True
>>> board
['--------', '---XOOOO', 'XXXXXXXX', '--------', '------XO', '--------',
'---XXOXO', '--------']
>>> add_piece(board, "O", 2)
You can't add a piece to a full column!
False
>>> board
['--------', '---XOOOO', 'XXXXXXXX', '--------', '------XO', '--------',
'---XXOXO', '--------']
>>> add_piece(board, "e", 1)
True
>>> board
['--------', '--eXOOOO', 'XXXXXXXX', '--------', '------XO', '--------',
'---XXOXO', '--------']
```

4.9 remove_piece(board: list[str], column_index: int) -> bool

Removes the bottom-most piece from the column at the given `column_index` (0-indexed) of the given `board` according to the game rules, and moves all other pieces in the relevant column down a row. If the requested column is empty, then a piece is not removed and a message is displayed to the user as described in Table 2. Returns True if a piece was removed from the board, and False otherwise. Note that this function mutates the given board and does *not* return a new board state.

A precondition to this function is that the specified board will contain exactly 8 strings each with exactly 8 characters, and represent a valid game state. Another precondition to this function is that the specified column index must be between 0 and 7 inclusive.
Example:

```
>>> board = ['--------', '----OOOO', 'XXOOOXXX', '--------', '------XO',
 '--------', '---XXOXO', '--------']
>>> remove_piece(board, 2)
True
>>> board
['--------', '----OOOO', '-XXOOOXX', '--------', '------XO', '--------',
 '---XXOXO', '--------']
>>> remove_piece(board, 0)
You can't remove a piece from an empty column!
False
>>> board
['--------', '----OOOO', '-XXOOOXX', '--------', '------XO', '--------',
 '---XXOXO', '--------']
```

4.10 check_win(board: list[str]) -> Optional[str]

Checks the given board state for a win or draw. If one player has formed an unbroken line (horizontal, vertical, or diagonal) of at least 4 of their own pieces, then this function returns that players piece. If both players have formed unbroken lines (horizontal, vertical, or diagonal) of at least 4 of their own pieces, then this function returns the blank piece. If neither player has formed an unbroken line (horizontal, vertical, or diagonal) of at least 4 of their own pieces, then this function returns None.
A precondition to this function is that the specified board will contain exactly 8 strings each with exactly 8 characters, all of which will be one of either `X`,`O`, or `-`. Example:

```
>>> board = ['------XO', '-------O', '--------', '--------', '-------O',
 '--------', '--------', '------XX']
>>> check_win(board)
>>> board = ['-------O', '------OX', '-----OXO', '---XOOXX', '--------',
 '--------', '--------', '--------']
>>> check_win(board)
'O'
>>> board = ['-------X', '-------X', '------OX', '---OOOXX', '--------',
 '--------', '--------', '--------']
>>> check_win(board)
'X'
>>> board = ['---XXXXO', '-------O', '-------O', '-------O', '--------',
 '--------', '--------', '--------']
>>> check_win(board)
'-'
>>> board = ['--------', '--------', '---O----', '---O----', '---O----',
 '---O----', '--------', '--------']
>>> check_win(board)
'O'
```

4.11 `play_game() -> None`

Coordinates gameplay of a single game from start to finish. This function should follow steps 1 to 7 (inclusive) presented in section 3. The `play_game` function should utilize other functions you have written. In order to make the `play_game` function shorter, you should consider writing extra helper functions.
The output from your `play_game` function (including prompts) must exactly match the expected output. Running the sample tests will give you a good idea of whether your prompts and other outputs are correct. Use samples of gameplay from the `gameplay/` folder provided with this assignment for examples of how the `play_game` function should run.

4.12 `main() -> None`

The main function should be called when the file is run. The main function enacts a game of connect 4 using the `play_game` function, and then follows step 8 presented in section 3.
The `gameplay/` folder provided with this assignment contains full gameplay examples which should demonstrate how the `main` function should run.
In the provided `a1.py`, the function definition for `main` has already been provided, and the `if __name__ == "__main__":` block will ensure that the code in the `main` function is run when your `a1.py` file is run. Do not call your `main` function outside of this block, and do not call any other function outside this block unless you are calling them from within the body of another function.

5 Assessment and Marking Criteria

This assignment assesses course learning objectives:

1. apply program constructs such as variables, selection, iteration and sub-routines,
2. read and analyse code written by others,
3. read and analyse a design and be able to translate the design into a working program, and
4. apply techniques for testing and debugging.

5.1 Functionality

Your program's functionality will be marked out of a total of 6 marks. Your assignment will be put through a series of tests and your functionality mark will be proportional to the number of tests you pass. You will be given a *subset* of the functionality tests before the due date for the assignment.

You may receive partial marks within each section for partially working functions, or for implementing only a few functions.

You need to perform your *own* testing of your program to make sure that it meets *all* specifications given in the assignment. Only relying on the provided tests is likely to result in your program failing in some cases and you losing some functionality marks. Note: Functionality tests are automated, so string outputs need to match *exactly* what is expected.

Your program must run in Gradescope, which uses Python 3.12. Partial solutions will be marked but if there are errors in your code that cause the interpreter to fail to execute your program, you will get zero for functionality marks. If there is a part of your code that causes the interpreter to fail, comment out the code so that the remainder can run. Your program must run using the Python 3.12 interpreter. If it runs in another environment (e.g. Python 3.8 or PyCharm) but not in the Python 3.12 interpreter, you will get zero for the functionality mark.

5.2 Code Style

The style of your assignment will be assessed by a tutor. Style will be marked according to the style rubric provided with the assignment. The style mark will be out of 4.

The key consideration in marking your code style is whether the code is easy to understand. There are several aspects of code style that contribute to how easy it is to understand code. In this assignment, your code style will be assessed against the following criteria.

- Readability
  - Program Structure: Layout of code makes it easy to read and follow its logic. This includes using whitespace to highlight blocks of logic.
  - Descriptive Identifier Names: Variable, constant, and function names clearly describe what they represent in the program's logic. Do not use Hungarian Notation for identifiers. In short, this means do not include the identifier's type in its name, rather make the name meaningful (e.g. employee identifier).
  - Named Constants: Any non-trivial fixed value (literal constant) in the code is represented by a descriptive named constant (identifier).
- Algorithmic Logic
  - Single Instance of Logic: Blocks of code should not be duplicated in your program. Any code that needs to be used multiple times should be implemented as a function.
  - Variable Scope: Variables should be declared locally in the function in which they are needed. Global variables should not be used.
  - Control Structures: Logic is structured simply and clearly through good use of control structures (e.g. loops and conditional statements).
- Documentation:
  - Comment Clarity: Comments provide meaningful descriptions of the code. They should not repeat what is already obvious by reading the code (e.g. `# Setting variable to 0`). Comments should not be verbose or excessive, as this can make it difficult to follow the code.
  - Informative Docstrings: Every function should have a docstring that summarises its purpose. This includes describing parameters and return values (including type information) so that others can understand how to use the function correctly.
  - Description of Logic: All significant blocks of code should have a comment to explain how the logic works. For a small function, this would usually be the docstring. For long or complex functions, there may be different blocks of code in the function. Each of these should have an in-line comment describing the logic.

5.3 Assignment Submission

You must submit your assignment electronically via Gradescope (`https://gradescope.com/`). You **must** use your UQ email address which is based on your student number (e.g. s4123456@student.uq.edu.au) as your Gradescope submission account.

When you login to Gradescope you may be presented with a list of courses. Select CSSE1001. You will see a list of assignments. Choose **Assignment 1**. You will be prompted to choose a file to upload. The prompt may say that you can upload any files, including zip files. You **must** submit your assignment as a single Python file called `a1.py` (use this name – all lower case), and *nothing* else. Your submission will be automatically run to determine the functionality mark. If you submit a file with a **different name**, the tests will **fail** and you will get **zero** for functionality. Do **not** submit **any** sort of archive file (e.g. zip, rar, 7z, etc.).

Upload an initial version of your assignment *at least* one week before the due date. Do this even if it is just the initial code provided with the assignment. If you are unable access Gradescope, contact the course helpdesk (csse1001@helpdesk.eait.uq.edu.au) *immediately.* Excuses, such as you were not able to login or were unable to upload a file will not be accepted as reasons for granting an extension.

When you upload your assignment it will run a **subset** of the functionality autograder tests on your submission. It will show you the results of these tests. It is your responsibility to ensure that your uploaded assignment file runs and that it passes the tests you expect it to pass.

Late submissions of the assignment will **not** be marked. Do not wait until the last minute to submit your assignment, as the time to upload it may make it late. Multiple submissions are allowed and encouraged, so ensure that you have submitted an almost complete version of the assignment *well* before the submission deadline of 16:00. Your latest, on time, submission will be marked. Ensure that you submit the correct version of your assignment.

In the event of exceptional personal or medical circumstances that prevent you from handing in the assignment on time, you may submit a request for an extension. See the course profile for details of how to apply for an extension.

Requests for extensions must be made **before** the submission deadline. The application and supporting documentation (e.g. medical certificate) must be submitted via my.UQ. You must retain the original documentation for a minimum period of six months to provide as verification, should you be requested to do so.

5.4 Plagiarism

This assignment must be your own individual work. By submitting the assignment, you are claiming it is entirely your own work. You **may** discuss general ideas about the solution approach with other students. Describing details of how you implement a function or sharing part of your code with another student is considered to be **collusion** and will be counted as plagiarism. You **may not** copy fragments of code that you find on the Internet to use in your assignment.

Please read the section in the course profile about plagiarism. You are encouraged to complete *both* parts A and B of the academic integrity modules *before* starting this assignment. Submitted assignments will be electronically checked for potential cases of plagiarism.