Fancy Naughts & Crosses

Assignment 1
Semester 2, 2023
CSSE1001/CSSE7030

Due date: 25 August 2023, 15:00 GMT+10

1 Introduction

In this assignment, you will implement a text-based modified naughts and crosses game inspired by this demo. The game is similar to regular naughts and crosses, but differs in that each player has markers of different sizes. The game is played on a 3×3 grid between two players. Starting with naughts (O), the players take turns placing their markers in cells on the board. A player may place any of their markers on an empty cell, or they may place a larger marker on top of a smaller marker already present on the board. The top-most marker is the one considered to be occupying the cell. The game is over when either:

1. One player *wins* by filling a row, column, or diagonal with their markers (note that the only marker that counts for the win check is the top-most marker)

2. The game reaches *stalemate* when no move can be made on the board but no player has met the win condition.

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.

`constants.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 import statements to `a1.py`. Doing so 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, and with both players having all available pieces. Naught (O) gets to make the first move. Until the end of the game, the following steps repeat:

1. The current game state is displayed (including the pieces remaining for each player and the current board state).
2. The user is informed whose turn it is to move.
3. The user is prompted for a move. At this prompt they may enter a valid move in the format `"{row} {column} {size}"`, or they may request to see a help message by entering `"h"` or `"H"`. If the user enters `"h"` or `"H"`, they should be shown the help message before being reprompted for a move. Otherwise, progress to the next step.
4. If the move is invalid for any reason, show the user a message to inform them of why their move was invalid (see Table 1 for all required validity checking and messages for this step), and then return to step 3. If the move is valid, progress to the next step.
5. Update the board according to the requested move and display the new game state.
6. If the game is not over, return to step 2.
7. If the game is over, inform the users of the outcome and prompt them for whether they would like to play again. At this prompt, if they enter either 'y' or 'Y', create a new game (i.e. set up an empty board and give the players all their pieces back) and return to step 1. If they enter anything other than 'y' or 'Y', terminate the program gracefully.

Note that we will not test the case where one player cannot make a move and the other player can; that is, you do not need to handle the case that the current player is out of moves to make but the next player would be able to make a move.

4 Implementation

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

Issue with user input	**Constant in `constants.py`**
The move entered is not 5 characters long, or does not contain 3 non-space characters each separated by a space character	`INVALID_FORMAT_MESSAGE`
The first character is not a valid row on the board	`INVALID_ROW_MESSAGE`
The second non-space character is not a valid column on the board	`INVALID_COLUMN_MESSAGE`
The final character is not a valid piece size (note that a piece size may still be valid even if the player has already played that piece). Valid piece sizes are between 1 and whatever value is stored in `PIECES_PER_PLAYER` inclusive.	`INVALID_SIZE_MESSAGE`
The move is requesting to place a marker the player has already placed, or is requesting to place a piece over a marker of the same or greater size	`INVALID_MOVE_MESSAGE`

Table 1: 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).

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.

The type hints in these functions make reference to `Board`, `Pieces`, and `Move` type aliases defined in the provided `a1.py`. A variable of type `Board` is of type `list[list[str]]`, meaning that it is a list where each element is also a list. The internal lists contain strings. A variable of type `Pieces` is of type `list[int]` meaning it is a list containing strings. See example usages of the relevant functions for what to expect in these variables. A `Move` is of type `tuple[int, int, int]`, meaning it is a tuple containing exactly 3 integers.

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*. 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 6.3 for instructions on how to submit your assignment to Gradescope.

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.

4.2 `generate_initial_pieces(num_pieces: int) -> Pieces`

Return a list of the initial marker sizes, from 1 up to and including `num_pieces`. You may assume that `num_pieces` will be between 5 and 9 inclusive. The piece sizes within a single players piece

list can be assumed to be unique throughout this assignment (that is, one player cannot have two pieces of the same size).

Example:

```
>>> generate_initial_pieces(5)
[1, 2, 3, 4, 5]
>>> generate_initial_pieces(8)
[1, 2, 3, 4, 5, 6, 7, 8]
```

4.3 initial_state() -> Board

Returns a new board where every cell (i.e. row, column position) contains `EMPTY`.

Example:

```
>>> initial_state()
[['  ', '  ', '  '], ['  ', '  ', '  '], ['  ', '  ', '  ']]
```

4.4 place_piece(board: Board, player: str, pieces_available: Pieces, move: Move) -> None

This function should place the requested piece at the requested position on the board and remove that piece from the `pieces_available` list. `move` is a tuple containing (row, column, piece size). That is, `move` specifies the requested (row, column) position on which to place `player`'s piece of the requested piece size. The marker placed should be a string of length 2, containing the `player` marker and piece size respectively. Note that this function does not return anything; its job is to *mutate* (i.e. change) the `board` and `pieces_available`.

You may assume that the requested piece size will be present in `pieces_available`.

This function does not need to handle checking for whether the move is valid according to the game rules.

Example:

```
>>> board = initial_state()
>>> pieces = generate_initial_pieces(6)
>>> board
[['  ', '  ', '  '], ['  ', '  ', '  '], ['  ', '  ', '  ']]
>>> pieces
[1, 2, 3, 4, 5, 6]
>>> place_piece(board, NAUGHT, pieces, (1, 1, 3))
>>> board
[['  ', '  ', '  '], ['  ', 'O3', '  '], ['  ', '  ', '  ']]
>>> pieces
[1, 2, 4, 5, 6]
>>> place_piece(board, NAUGHT, pieces, (1, 1, 1))
>>> board
[['  ', '  ', '  '], ['  ', 'O1', '  '], ['  ', '  ', '  ']]
>>> pieces
[2, 4, 5, 6]
```

4.5 print_game(board: Board, naught_pieces: Pieces, cross_pieces: Pieces) -> None

Displays the game in a user-friendly format. Your output must exactly match the expected formatting (including whitespace and grammar) in order to receive marks. Ensure when testing with the below examples that your output exactly matches the output shown. The authority on the correct formatting are the Gradescope tests. Ensure your output aligns exactly with the Gradescope sample tests.

Example:

```
>>> board = initial_state()
>>> naught_pieces = generate_initial_pieces(5)
>>> cross_pieces = generate_initial_pieces(5)
>>> print_game(board, naught_pieces, cross_pieces)
O has: 1, 2, 3, 4, 5
X has: 1, 2, 3, 4, 5

   1  2  3
  ---------
1|  |  |  |
  ---------
2|  |  |  |
  ---------
3|  |  |  |
  ---------
>>> place_piece(board, NAUGHT, naught_pieces, (1, 1, 1))
>>> place_piece(board, CROSS, cross_pieces, (0, 0, 1))
>>> print_game(board, naught_pieces, cross_pieces)
O has: 2, 3, 4, 5
X has: 2, 3, 4, 5

   1  2  3
  ---------
1|X1|  |  |
  ---------
2|  |O1|  |
  ---------
3|  |  |  |
  ---------
```

4.6 process_move(move: str) -> Move | None

Attempts to convert the move string to a tuple of three integers representing the (row, column, piece size) of the move. If the move is invalid, this function should print the relevant message and return None; see the first 4 rows of Table 1 for the relevant messages. Note that this function does not need to handle the final issue described in Table 1. If the move is in the valid format, this function returns a tuple containing 3 integers, representing the row, column, and size of the piece respectively.

If the conversion is successful, this function should return the converted move. Note that, in the string move, the row and column will be 1-indexed, whereas in the returned tuple the row and

column should be 0-indexed.

Example:

```
>>> process_move('apple')
Invalid move format. Please try again.

>>> process_move('apple orange berry')
Invalid move format. Please try again.

>>> process_move('a o b')
Invalid row. Please try again.

>>> process_move('0 0 1')
Invalid row. Please try again.

>>> process_move('1 0 1')
Invalid column. Please try again.

>>> process_move('1 1 a')
Invalid piece size. Please try again.

>>> process_move('1 1 1')
(0, 0, 1)
>>> process_move('3 3 5')
(2, 2, 5)
>>> result = process_move('apple')
Invalid move format. Please try again.

>>> result
>>> result = process_move('2 2 2')
>>> result
(1, 1, 2)
```

4.7 get_player_move() -> Move

Continuously prompts the user for a move until a move is entered in a valid format, and returns information about the final valid move entered as a tuple of integers in the format (row, column, size). This function should handle displaying a help message if the user enters either `"h"` or `"H"` at the prompt (and then continue prompting until a valid move is entered). Each time the user enters something in an invalid format, the relevant message should be displayed as per the first four rows of Table 1. Note that the final row does not need to be handled in this function.

Example:

```
>>> get_player_move()
Enter your move: apple orange banana
Invalid move format. Please try again.

Enter your move: 0 0 0
Invalid row. Please try again.
```

```
Enter your move: 1 1 0
Invalid piece size. Please try again.

Enter your move: 1 1 1
(0, 0, 1)
>>> get_player_move()
Enter your move: h
Enter a row, column & piece size in the format: row col size

Enter your move: 2 2 2
(1, 1, 2)
```

4.8 check_move(board: Board, pieces_available: Pieces, move: Move) -> bool

Checks whether `move` is a valid move for the current `board` according to the game rules. In order to be a valid move, the piece size must be available in the given `pieces_available` list, and the (row, column) position described in `move` must be empty, or contain a piece of size smaller than that described in `move`. Returns `True` if the move is valid and `False` otherwise.

You may assume within this function that the `move` is within the bounds of the board and refers to a piece size that would have existed in the *initial* pieces for the player.

4.9 check_win(board: Board) -> str | None

Checks whether the game has been won. If the game has been won, this function should return who won. Otherwise, this function should return None.

Example:

```
>>> board = [['O1', 'O2', 'O3'], [EMPTY, EMPTY, EMPTY], [EMPTY, EMPTY, EMPTY]]
>>> check_win(board)
'O'
>>> board = initial_state()
>>> check_win(board)
>>> board = [['X1', 'O2', EMPTY], [EMPTY, 'X2', 'O3'], ['O4', EMPTY, 'X8']]
>>> check_win(board)
'X'
```

4.10 check_stalemate(board: Board, naught_pieces: Pieces, cross_pieces: Pieces) -> bool

Returns `True` if there are no more moves that can be made in this game, else `False`. Note that a player can place a marker on top of one of their own markers, provided the existing marker is smaller than the new marker.

Example:

```
>>> board = [['X3', 'O3', 'X2'], ['O4', 'X4', 'O2'], ['O5', 'X6', 'O6']]
>>> check_win(board)
>>> check_stalemate(board, [1], [1])
True
```

```
>>> check_stalemate(board, [1], [1, 5])
False
```

4.11 main() -> None

The main function should be called when the file is run, and coordinates the overall gameplay. Section 3 describes how the game should be played. The `main` function should utilize other functions you have written. In order to make the `main` function shorter, you should consider writing extra helper functions. 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. The output from your `main` 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. See the `gameplay/` folder provided with this assignment for examples of how the `main` function should run.

5 CSSE7030 Task: Changing GRID_SIZE

This section describes an additional task that CSSE7030 students are required to complete for full marks. Students in CSSE1001 do not need to attempt this task, and cannot earn marks for it.

For this task, ensure your program works when `GRID_SIZE` is changed in `constants.py` to a different value. We will only test your code with `GRID_SIZE` values in the range $[2, 8]$. However, you ideally should not hardcode your solution to only work for values in the range 2 to 8; these are simply provided as guarantees for the range of values we will test within. A well written solution would likely generalize to other grid sizes. For examples of how output should update for different grid sizes, please see the `7030_example_9_pieces_6_grid_size.py` and `7030_example_5_pieces_2_grid_size.py` files in `gameplay/`.

You may assume that the grid is always square (i.e. `GRID_SIZE` describes both the number of rows and the number of columns in the grid).

6 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.

6.1 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 3.11. 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 3.11 interpreter. If it runs in another environment (e.g. Python 3.8 or PyCharm) but not in the Python 3.11 interpreter, you will get zero for the functionality mark.

6.2 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.

6.3 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/CSSE7030. 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 15:00. 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.

6.4 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.