Battleships

Assignment 1
Semester 2, 2024
CSSE1001

Due date: 23rd August 2024, 15:00 GMT+10

1 Introduction

In Assignment 1 you will implement a text-based version of the game *Battleships*. This two player game consists of two phases. Firstly, players secretly place some number of *ships* onto their own *board* which is hidden from their opponent. Then, players take turns guessing positions on their opponents board in an attempt to *hit* parts of their opponents ships. A player wins the game once they have hit all positions occupied by their opponents ships. You can find instructions for how this game is traditionally played here. Section 3 provides an overview of gameplay for the CSSE1001 assignment. Where the original behaviour of the game conflicts with this document, you should implement your program according to this document.

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.

`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.

3 Gameplay

This section provides a high-level overview of gameplay, and is intended to provide you with an idea of the behaviour of a fully completed assignment 1. Do not simply implement the behaviour described here using your own structure; you must implement the individual functions described in Section 4. Prompts and outputs must match *exactly* what is expected; see Section 4, the example games in the `gameplay/` folder provided with this assignment, and the Gradescope tests for clarification on required prompts and outputs.

3.1 Setup Phase

At the beginning of the game the following steps should occur:

1. Users should be prompted for the board size to use (you may assume the user will enter a number between 2 and 9 inclusive at this prompt).
2. Users should be prompted to enter a comma-separated string of whole numbers that will dictate the ship sizes both players will use (you may assume the user will enter a valid comma-separated string of whole numbers at this prompt).
3. Each user should be repeatedly prompted in turn to set up their boards by placing ships of the required sizes. If a user enters an invalid ship at any point they should be reprompted.

Once both players have placed their ships, they move to the turn-taking phase.

3.2 Turn-taking Phase

Players alternate turns, starting with player 1. During a single turn, the following steps should occur:

1. If a player has won, the following information should be displayed in order: a message showing that the game is over, a message stating which player won, and the final game state with all ships depicted. The program should then terminate gracefully. Steps 2-7 would therefore not execute if a player has won.
2. A message should be displayed to show that a new turn has begun.
3. The current game state should be displayed, *without* ships shown.
4. A message should be displayed to indicate whose turn it is.
5. The player should be repeatedly prompted to make an attack until they enter a valid attack.
6. The specified valid attack should be made.
7. Player turn should alternate.

4 Implementation

This assessment 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) week 4. Namely, the following techniques are permitted for use in this assignment:

- Functions (`def`, `return`)
- Basic control structures (`for`, `while`, `if`/`elif`/`else`, `break`)
- Primitive data types (`int`, `float`, `str`, `bool`)
- Variable assignment (`=`)
- Arithmetic (`+`,`-`,`*`,`\`,`\\`, ,`%` etc.)
- Comparison (`==`,`<=`,`>=`,`<`,`>`,`!=` etc.)

- Basic Logic (`not`, `and`, `or` etc.)
- `lists` and `tuples`
- `dictionaries`
- `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) week 4 will result in a deduction of up to 100% of your mark.

A pinned thread will be maintained on the Edstem 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 (as a *float*) that you have spent on the assignment. The purpose of this function is to enable you to verify that you understand how to submit to Gradescope as soon as possible (see Section 5.3), 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. 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.

4.2 `create_empty_board(board_size: int) -> list[str]`

Generates a new empty board. A board is represented by a list of strings, where each string in the list represents a row on the board. The first string in the list represents top-most row and the last string in the list represents the bottom-most row. The first character of each string represents the square at the left-most column of that row and the last character of the string represents the right-most column of the row. The board should have `board_size` rows and `board_size` columns. A pre-condition to this function is that `board_size` must be between 2 and 9 inclusive.

```
>>> create_empty_board(4)
['~~~~', '~~~~', '~~~~', '~~~~']
>>> create_empty_board(6)
['~~~~~~', '~~~~~~', '~~~~~~', '~~~~~~', '~~~~~~', '~~~~~~']
```

4.3 `get_square(board: list[str], position: tuple[int,int]) -> str`

Returns the character present at the given (row, column) `position` within the given `board`. A precondition to this function is that `position` must exist on `board`.

```
>>> board = ['~~~!', '!O~~', '~~~~', '~~~~']
>>> get_square(board, (1, 0))
'!'
>>> get_square(board, (0, 1))
'~'
>>> get_square(['abc', 'de'], (1, 1))
'e'
```

4.4 `change_square(board: list[str], position: tuple[int, int], new_square: str) -> None`

Replaces the character at the given (row, column) `position` with `new_square`. Note that this function should *mutate* the given `board` and *not* return a new board state. A precondition to this function is that `position` must exist on `board`.

```
>>> board = create_empty_board(5)
>>> board
['~~~~~', '~~~~~', '~~~~~', '~~~~~', '~~~~~']
>>> change_square(board, (2, 3), MISS_SQUARE)
>>> board
['~~~~~', '~~~~~', '~~~!~', '~~~~~', '~~~~~']
>>> board = ['abc', 'de']
>>> change_square(board, (1, 1), 'h')
>>> board
['abc', 'dh']
```

4.5 `coordinate_to_position(coordinate: str) -> tuple[int, int]`

Returns the (row, column) position tuple corresponding to the given `coordinate`. A `coordinate` is a string entered by a user to represent a (row, column) position on the board. Coordinates consist of exactly two characters, where the first character is an uppercase letter representing the column and the second character is a digit representing the row. Letters that appear earlier in the alphabet represent lower column numbers (e.g. 'A' represents column 0, 'B' represents column 1, etc.). The digit character representing the row number is 1-indexed, whereas the actual row it

represents is 0-indexed (e.g. '1' represents row 0, '2' represents row 1, etc.). A precondition to this function is that the coordinate must consist of exactly two characters, where the first character is an uppercase letter from 'A' to 'I' and the second character is a single digit character.

```
>>> coordinate_to_position('A1')
(0, 0)
>>> coordinate_to_position('B3')
(2, 1)
>>> coordinate_to_position('G8')
(7, 6)
```

4.6 can_place_ship(board: list[str], ship: list[tuple[int,int]]) -> bool

Returns True if the `ship` can be placed on the `board`, and False otherwise. A ship can be placed on a board if and only if all squares it intends to occupy are empty. A precondition to this function is that all positions in `ship` must exist on `board`.

```
>>> board = ['~~~~', 'OO~~', '~~~~', '~~~~']
>>> can_place_ship(board, [(0, 0), (0, 1)])
True
>>> board
['~~~~', 'OO~~', '~~~~', '~~~~']
>>> can_place_ship(board, [(0, 0), (1, 0)])
False
```

4.7 place_ship(board: list[str], ship: list[tuple[int,int]]) -> None

Places the `ship` on the `board` by changing all positions from the ship to the `ACTIVE_SHIP_SQUARE`. A precondition to this function is that the `ship` should be able to be placed on the `board` according to `can_place_ship`.

```
>>> board = ['~~~~', 'OO~~', '~~~~', '~~~~']
>>> place_ship(board, [(0, 0), (0, 1)])
>>> board
['OO~~', 'OO~~', '~~~~', '~~~~']
```

4.8 attack(board: list[str], position: tuple[int, int]) -> None

Attempts to attack the cell at the (row, column) `position` within the `board`. If the `position` contains an active ship square, the attack *hits* and converts this square to a dead ship square. If the `position` contains an empty square, then the attack *misses* and converts this square to a miss square. If the `position` contains anything else, this function should not change `board`. A precondition to this function is that `position` must exist on `board`.

```
>>> board = ['O~~~', 'O~~~', '~~~~', '~~~~']
>>> attack(board, (0, 0))
>>> board
['X~~~', 'O~~~', '~~~~', '~~~~']
>>> attack(board, (0, 1))
>>> board
['X!~~', 'O~~~', '~~~~', '~~~~']
>>> attack(board, (0, 0))
```

```
>>> board
['X!~~', 'O~~~', '~~~~', '~~~~']
```

4.9 display_board(board: list[str], show_ships: bool) -> None

Prints the board in a human-readable format. If `show_ships` is `False`, all active ship squares should be displayed as empty squares. If `show_ships` is `True`, all squares should be shown as they are (including active ship squares).

```
>>> board = create_empty_board(4)
>>> display_board(board, True)
 /ABCD
1|~~~~
2|~~~~
3|~~~~
4|~~~~
>>> place_ship(board, [(0, 0), (1, 0)])
>>> attack(board, (0, 0))
>>> display_board(board, False)
 /ABCD
1|X~~~
2|~~~~
3|~~~~
4|~~~~
>>> display_board(board, True)
 /ABCD
1|X~~~
2|O~~~
3|~~~~
4|~~~~
>>> board = create_empty_board(8)
>>> display_board(board, False)
 /ABCDEFGH
1|~~~~~~~~
2|~~~~~~~~
3|~~~~~~~~
4|~~~~~~~~
5|~~~~~~~~
6|~~~~~~~~
7|~~~~~~~~
8|~~~~~~~~
```

4.10 get_player_hp(board: list[str]) -> int

Returns the players current HP, which is equal to the total number of active ship squares remaining in their board.

```
>>> board = ['O~~~', 'O~~~', '~~~~', '~~~~']
>>> get_player_hp(board)
2
>>> attack(board, (0, 0))
>>> get_player_hp(board)
1
```

4.11 display_game(p1_board: list[str], p2_board: list[str], show_ships: bool) -> None

Prints the overall game state. The game state consists of player 1's health and board state, followed by player 2's health and board state. `show_ships` determines whether ships should be shown when displaying the players' boards.

```
>>> player_1_board = ['O~~~', 'O~~~', '~~~~', '~~~~']
>>> player_2_board = ['~~~~', '~~~~', 'OO~~', '~~~~']
>>> attack(player_1_board, (0, 0))
>>> attack(player_2_board, (0, 0))
>>> display_game(player_1_board, player_2_board, True)
PLAYER 1: 1 life remaining
 /ABCD
1|X~~~
2|O~~~
3|~~~~
4|~~~~
PLAYER 2: 2 lives remaining
 /ABCD
1|!~~~
2|~~~~
3|OO~~
4|~~~~
>>> display_game(player_1_board, player_2_board, False)
PLAYER 1: 1 life remaining
 /ABCD
1|X~~~
2|~~~~
3|~~~~
4|~~~~
PLAYER 2: 2 lives remaining
 /ABCD
1|!~~~
2|~~~~
3|~~~~
4|~~~~
```

4.12 is_valid_coordinate(coordinate: str, board_size: int) -> tuple[bool, str]

Checks if the provided `coordinate` represents a valid coordinate string. This function should return a tuple containing a boolean (which is True if the coordinate is valid and False otherwise), and a string containing a message to describe the issue (if any) with the `coordinate`. Table 1 describes the possible issues that `coordinate` can have, and the corresponding messages that should be returned.

```
>>> is_valid_coordinate('A3', 4)
(True, '')
>>> is_valid_coordinate('A4', 3)
(False, 'Invalid coordinate number.')
>>> is_valid_coordinate('abcd', 4)
```

Issue	Message
Coordinate does not contain exactly two characters	`INVALID_COORDINATE_LENGTH`
First character in coordinate does not represent a valid column letter for a board with the given `board_size`	`INVALID_COORDINATE_LETTER`
Second character in coordinate does not represent a valid row digit for a board with the given `board_size`	`INVALID_COORDINATE_NUMBER`
No issue (none of the issues described above are present)	Empty string (`''`)

Table 1: Messages to be included in the returned tuple, based on the issue present in `coordinate`. Precedence is top down; that is, if there are multiple issues with a coordinate, the one higher in the table is the one that should be used. The constants present in this table are defined in `support.py`.

```
(False, 'Coordinates should be 2 characters long.')
>>> is_valid_coordinate('AA', 4)
(False, 'Invalid coordinate number.')
>>> is_valid_coordinate('H3', 5)
(False, 'Invalid coordinate letter.')
```

4.13 `is_valid_coordinate_sequence(coordinate_sequence: str, ship_length: int, board_size: int) -> tuple[bool, str]`

Checks if the provided `coordinate_sequence` represents a sequence of exactly `ship_length` comma-separated valid coordinate strings. This function should return a tuple containing a boolean (which is True if there are exactly `ship_length` coordinates which are all valid, and False otherwise), and a string containing a message to describe the issue (if any) with the `coordinate_sequence`. If the coordinate sequence does not contain exactly `ship_length` comma-separated *items*, the message returned should be `INVALID_COORDINATE_SEQUENCE_LENGTH`. Otherwise, each item should be checked for validity in order. At the first invalid coordinate, this function should return a tuple containing False and the string message describing the issue with that coordinate. If all items are valid coordinates, this function should return a tuple containing True and the empty string. Note that cases of ships being invalid due to reasons other than those described in this document (e.g. occupying the same square twice, positions disconnected from other positions within the same ship, etc.) will not be tested.

```
>>> is_valid_coordinate_sequence('A1,B2,C3', 3, 4)
(True, '')
>>> is_valid_coordinate_sequence('A1,B2,C3', 3, 2)
(False, 'Invalid coordinate letter.')
>>> is_valid_coordinate_sequence('A1,B2,C3', 2, 2)
(False, 'Invalid coordinate sequence length.')
>>> is_valid_coordinate_sequence('A1,B2,ABCD', 3, 4)
(False, 'Coordinates should be 2 characters long.')
>>> is_valid_coordinate_sequence('E8,B2,ABCD', 3, 4)
(False, 'Invalid coordinate letter.')
```

4.14 `build_ship(coordinate_sequence: str) -> list[tuple[int, int]]`

Returns the list of (row, column) positions corresponding to the `coordinate_sequence`. The order of the returned positions must match the order of the given coordinates. A precondition to this function is that `coordinate_sequence` must represent a valid coordinate sequence.

```
>>> build_ship('A1,A2,A3')
[(0, 0), (1, 0), (2, 0)]
>>> build_ship('G4,G5,G6,G7,G8')
[(3, 6), (4, 6), (5, 6), (6, 6), (7, 6)]
```

4.15 setup_board(board_size: int, ship_sizes: list[int]) -> list[str]

Allows the user to set up a new board by placing ships. This function should create a new board with the given `board_size`, and then prompt the user for a ship of each of the given `ship_sizes` in turn. Each time the user is prompted, the board state should first be displayed (with ships shown), before the appropriate prompt is displayed. If the user enters an invalid coordinate sequence at any point, the issue with the coordinate sequence should be displayed and the user should be reprompted with the same ship size. Each time the user enters a valid coordinate sequence, the corresponding ship should be constructed. If the ship cannot be placed on the board (see `can_place_ship`), the `INVALID_SHIP_PLACEMENT` message should be displayed and the user should be reprompted with the same ship size. Otherwise, the ship should be placed onto the board. Once ships of all required sizes have successfully been placed, this function should return the board.

```
>>> board = setup_board(4, [2, 3])
 /ABCD
1|~~~~
2|~~~~
3|~~~~
4|~~~~
Enter a comma separated list of 2 coordinates: A1,A2
 /ABCD
1|O~~~
2|O~~~
3|~~~~
4|~~~~
Enter a comma separated list of 3 coordinates: A1,A2,A3
Ship placement would overlap another ship.
 /ABCD
1|O~~~
2|O~~~
3|~~~~
4|~~~~
Enter a comma separated list of 3 coordinates: aaaaa
Invalid coordinate sequence length.
 /ABCD
1|O~~~
2|O~~~
3|~~~~
4|~~~~
Enter a comma separated list of 3 coordinates: C1,D1,E1
Invalid coordinate letter.
 /ABCD
1|O~~~
2|O~~~
3|~~~~
4|~~~~
```

```
Enter a comma separated list of 3 coordinates: B1,C1,D1
>>> board
['OOOO', 'O~~~', '~~~~', '~~~~']
```

4.16 get_winner(p1_board: list[str], p2_board: list[str]) -> Optional[str]

Returns which player has won (their opponent's hp is 0), or None if neither player has won yet. The winning players should be represented using the `PLAYER_ONE` and `PLAYER_TWO` constants in `support.py`. Note that it is not possible for a stalemate to occur; this function will only be tested with cases where at least one board has remaining active ship squares.

```
>>> p1_board = ['OOOO', 'O~~~', '~~~~', '~~~~']
>>> get_winner(p1_board, p1_board)
>>> p2_board = ['XX~~', '!~~~', 'XXX~', '!!!!']
>>> get_winner(p1_board, p2_board)
'PLAYER 1'
>>> get_winner(p2_board, p1_board)
'PLAYER 2'
```

4.17 make_attack(target_board: list[str]) -> None

Performs a single turn against the `target_board`. This involves prompting for a turn, ensuring the entered text is a valid coordinate, and attacking that position on the `target_board`. Each time an invalid coordinate is entered, a corresponding message should be displayed as per Table 1 and the user should be reprompted.

```
>>> board = ['OOOO', 'O~~~', '~~~~', '~~~~']
>>> make_attack(board)
Enter a coordinate to attack: AA
Invalid coordinate number.
Enter a coordinate to attack: abc
Coordinates should be 2 characters long.
Enter a coordinate to attack: A5
Invalid coordinate number.
Enter a coordinate to attack: A1
>>> board
['XOOO', 'O~~~', '~~~~', '~~~~']
>>> make_attack(board)
Enter a coordinate to attack: A1
>>> board
['XOOO', 'O~~~', '~~~~', '~~~~']
```

4.18 play_game() -> None

Coordinates gameplay of a single game of *Battleships* from start to finish according to section 3. The printed output from your `play_game` function (including prompts) must exactly match the expected output. Running the Gradescope tests will give you a good idea of whether your prompts and other outputs are correct. The `gameplay/` folder provided with this assignment contains full gameplay examples which should demonstrate how the `play_game` function should run.

In the provided `a1.py`, the function definition for `play_game` has already been provided, and the `if __name__ == "__main__":` block will ensure that the code in the `play_game` function is run when your `a1.py` file is run. Do not call your `play_game` 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.

The assignment will be marked out of 10, with 6 points available for functionality and 4 points available for style.

5.1 Functionality

Your program's functionality will be marked automatically 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 with the release of this assignment. You can check if your code passes these functionality tests by uploading it to Gradescope. Note: Functionality tests are automated, so string outputs need to match *exactly* what is expected. You should also 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 may result in your program failing in some cases and you losing some functionality marks. If your final submission does not pass a test (even if this is due to a 'minor' reason such as a missing space) you will not be awarded the marks for that test.

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

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

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 Style

Your program's style will be marked manually out of a total of 4 marks. You will not receive these marks until a few weeks after the deadline for the assignment. The criteria used to mark your style are:

- 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, make a post on Edstem *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 submission of the assignment will result in a deduction of 100% of the total possible mark. A one-hour grace period will be applied to the due time, after which time (16:00) your submission

will be considered officially late and will receive a mark of 0. 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 submission will be marked. Do not submit after the deadline, as this will result in a late penalty of 100% of the maximum possible mark being applied to your submission.

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 **must not** copy fragments of code that you find on the Internet to use in your assignment. You **must not** use any artificial intelligence programs to assist you in writing 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.