# Assignment 1: Teeko

Introduction to Programming (CSSE7030)  
****Due: 12 September 2025, 15:00 GMT+10****

Published

August 14, 2025

## Contents

* [Changelog](https://csse7030.github.io/2025s2/a1" \l "changelog)
* [Introduction](https://csse7030.github.io/2025s2/a1" \l "introduction)
* [Getting Started](https://csse7030.github.io/2025s2/a1" \l "getting-started)
* [Gameplay](https://csse7030.github.io/2025s2/a1" \l "gameplay)
* [Implementaion](https://csse7030.github.io/2025s2/a1" \l "implementaion)
  + [Required Functions](https://csse7030.github.io/2025s2/a1" \l "required-functionsclasses-and-methods)
* [Assessment and Marking Criteria](https://csse7030.github.io/2025s2/a1" \l "assessment-and-marking-criteria)
  + [Functionality](https://csse7030.github.io/2025s2/a1" \l "functionality)
  + [Code Style](https://csse7030.github.io/2025s2/a1" \l "code-style)
  + [Assignment Submission](https://csse7030.github.io/2025s2/a1" \l "assignment-submission)
  + [Plagiarism and Referencing AI Use](https://csse7030.github.io/2025s2/a1" \l "plagiarism-and-referencing-ai-use)

## Changelog

* 15/08/25:
  + Initial Release.

## Introduction

Teeko is a 2 player strategy board game invented by John Scarne in 1937. In this game, players first take turns placing pieces on a 5x5 game board. Players then continue to take turns moving these pieces with the goal of being the first player to line their pieces up in either an unbroken line (horizontal, vertical, or diagonal) or a perfect square. [This website](https://teeko.cc/) gives a demonstration of the basic rules. You will be implementing a text based version of Teeko in this assignment. Note that this document takes precedence if there is any difference between it and the behavior of the above website. Instead of using colors, in our text based version of the game players will be represented using special symbols.

## Getting Started

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

* a1.py: This is the only file where you write your code and is the only .py file you will submit. 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.

## 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. Where interactions are not explicitly mentioned in this section, please see the [Implementation section](https://csse7030.github.io/2025s2/a1" \l "implementaion).

**Important**

Do not simply implement the behaviour described here using your own structure; you must implement the individual functions described in the [Implementation section](https://csse7030.github.io/2025s2/a1" \l "implementaion). Inputs and outputs must match exactly what is expected. Refer to the Implementation section, the given examples, and the provided [Gradescope tests](https://csse7030.github.io/2025s2/a1" \l "assignment-submission) for clarification on required prompts and outputs.

Gameplay takes place on a square board partitioned into positions. Players can place pieces on the board at empty positions, and move these pieces from one position to an adjacent empty position. Adjacent positions can either be horizontal, vertical, or diagonal. Players cannot place or move their opponent’s pieces. Player one’s pieces are represented using the special symbol ◍, and player two’s pieces are represented using the special symbol ●. These symbols are provided as constants in support.py and may appear slightly different on different operating systems such as Mac or Windows. Each Player has a total of four pieces. On a given player’s turn, they are not allowed to move any pieces unless all four of their pieces are already on the board. Players also cannot place another piece on the board on their turn if all four pieces are already placed. The game ends when one player has arranged their pieces in either an unbroken line (horizontal, vertical, or diagonal), or a perfect square. When this happens, the player that managed to arrange their pieces in this way is declared the winner.

The game begins with an empty board consisting of 5 rows and 5 columns. At the start of a game, a message is displayed welcoming both players. Players then alternate turns, starting with player one. A turn consists of the following steps:

1. The current game state is displayed.
2. A message is displayed indicating whose turn it is and whether they need to place or move a piece. If the player still has pieces to place, the number of remaining pieces is also shown.
3. The player is prompted to enter a command.
4. If the player entered a valid command, then action is taken according to [Table 1](https://csse7030.github.io/2025s2/a1" \l "tbl-actions). Otherwise the player is informed their command was invalid and gameplay returns to step 3.
5. The game state is checked. If a player has won, the state of the game and a message are displayed announcing their victory, and the game immediately proceeds to Step 7.
6. Player turn alternates and gameplay returns to step 1.
7. A prompt is displayed asking whether another game is desired.
8. If the user enters Y or y, the game is reset to the initial state described above and a new game begins, otherwise the program terminates gracefully.

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, the player should be informed their command was invalid, and the program should return immediately to step 3.

| **Valid Command** | **Action to take** |
| --- | --- |
| p[X][Y] or P[X][Y], Where [X] and [Y] are each single digit integers between 1 and 5 (inclusive) | If the current player has pieces left to place, attempt to place one of their pieces on the board at the (row, column) position (X,Y). Otherwise inform the player they have already placed all their pieces. If an attempt is made to place a piece at a position that is not empty, no piece is placed, and the player is informed with the appropriate message from support.py. If no piece is placed, gameplay returns to step 2. |
| m[X][Y][U][V] or M[X][Y][U][V], Where [X], [Y], [U], and [V] are each single digit integers between 1 and 5 (inclusive) | If the current player has no pieces left to place, attempt to move the piece at (row, column) position (X,Y) to (row, column) position (U,V). Otherwise inform the player they have pieces left to place. If an attempt is made to move an opponents piece, move a piece from an empty position, move a piece to an occupied position, or move a piece to a position that is not adjacent to where it was (horizontally, vertically, or diagonally); no piece is moved, and the player is informed with the appropriate message from support.py. If no piece is moved, gameplay returns to step 2. |
| H or h | A help message is displayed, and gameplay returns to step 2. |
| Q or q | The game is over and gameplay advances immediately to step 7. |

## Implementaion

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) Topic 5A (Dictionaries). 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) and their methods
* Type casting
* Variable assignment (=)
* Arithmetic (+,-,\*,\,\\, ,% etc.)
* Comparison (==,<=,>=,<,>,!= etc.)
* Basic Logic (not, and, or etc.)
* lists, tuples, and their methods
* dictionaries and their methods
* 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 5A (Dictionaries) will result in a deduction of up to 100% of your mark.

A pinned thread will be maintained on the [Edstem discussion board](https://edstem.org/au/courses/24510) 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.

**Important**

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 import statements already provided to you in a1.py. Removing or modifying these existing import statements may result in your code not functioning, and may result in a deduction of up to 100% of your mark.

### Required Functions

This section outlines the functions you are ****required**** to implement in a1.py. 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 zero marks for that test.

**Note**

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.

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.

You should be comfortable with course content up to and including the introduction of functions before commencing.

#### *Task 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 [Assignment Submission](https://csse7030.github.io/2025s2/a1" \l "assignment-submission)), 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.

Example:

>>> num\_hours()

10.0

#### *Task 2 create\_empty\_board(board\_size: int) -> list[list[str]]*

Generates an empty board with a specified number of rows. The generated board will be square (The number of columns will equal the number of rows). Empty board positions are marked with the special symbol ○ (See the EMPTY constant in support.py).

A board is represented by a list containing lists of strings. The first list in the overarching list of lists represents the top-most row, and the last list represents the bottom-most row. The first string of each row represents the left-most column of that row, and the last string of each row represents the right-most column of that row.

Preconditions:

* board\_size will not be negative

Example:

**Note**

In this and future examples, our assignment rendering software may display each row of the board on a new line. IDLE is unlikely to do this by default, but that is OK; it will not impact your grade.

>>> create\_empty\_board(0)

[]

>>> create\_empty\_board(1)

[['○']]

>>> create\_empty\_board(8)

[['○', '○', '○', '○', '○', '○', '○', '○'],

['○', '○', '○', '○', '○', '○', '○', '○'],

['○', '○', '○', '○', '○', '○', '○', '○'],

['○', '○', '○', '○', '○', '○', '○', '○'],

['○', '○', '○', '○', '○', '○', '○', '○'],

['○', '○', '○', '○', '○', '○', '○', '○'],

['○', '○', '○', '○', '○', '○', '○', '○'],

['○', '○', '○', '○', '○', '○', '○', '○']]

#### *Task 3 display\_board(board: list[list[str]]) -> None*

Prints the given game state to the screen in a visually appealing format. See the below examples for the correct format. Note that Gradescope requires your output to match exactly (including whitespace). Note that you are not expected to handle the misalignment caused by a number of rows or columns greater than 9.

Preconditions:

* board will contain at least one row and one column
* Each row of board will contain the same number of columns

**Tip**

This function does not return anything.

Example:

>>> board = create\_empty\_board(9)>>> display\_board(board)

1 2 3 4 5 6 7 8 9

1 ○ ○ ○ ○ ○ ○ ○ ○ ○

2 ○ ○ ○ ○ ○ ○ ○ ○ ○

3 ○ ○ ○ ○ ○ ○ ○ ○ ○

4 ○ ○ ○ ○ ○ ○ ○ ○ ○

5 ○ ○ ○ ○ ○ ○ ○ ○ ○

6 ○ ○ ○ ○ ○ ○ ○ ○ ○

7 ○ ○ ○ ○ ○ ○ ○ ○ ○

8 ○ ○ ○ ○ ○ ○ ○ ○ ○

9 ○ ○ ○ ○ ○ ○ ○ ○ ○

>>> board = [["!"]]>>> display\_board(board)

1

1 !

>>> board = [["A","B","C"],["D","E","F"],["G","H","I"]]>>> display\_board(board)

1 2 3

1 A B C

2 D E F

3 G H I

>>> board = [["A","N","D"],["R","E","W",],["A","Z","I"],["N","1","2"],["3","4","5"]]>>> display\_board(board)

1 2 3

1 A N D

2 R E W

3 A Z I

4 N 1 2

5 3 4 5

#### *Task 4 add\_piece(board: list[list[str]], piece: str, pos: tuple[int, int]) -> bool*

Adds the specified piece to the board at the given location if valid to do so.

Positions are given as (row, column) and are 1-Indexed (That is, the top left corner is given by position (1,1)). If the given position is not empty, the given piece is not added to the board and INVALID\_PLACEMENT\_MESSAGE given in support.py will be displayed. This function should return a boolean that is True if (and only if) a piece was successfully added to the board; otherwise, it should return False.

**Tip**

This function does not return a board, instead it mutates the given board state.

Preconditions:

* pos corresponds to a position that exists within board

Example:

>>> board = create\_empty\_board(5)>>> display\_board(board)

1 2 3 4 5

1 ○ ○ ○ ○ ○

2 ○ ○ ○ ○ ○

3 ○ ○ ○ ○ ○

4 ○ ○ ○ ○ ○

5 ○ ○ ○ ○ ○

>>> add\_piece(board, "X", (1,1))

True

>>> display\_board(board)

1 2 3 4 5

1 X ○ ○ ○ ○

2 ○ ○ ○ ○ ○

3 ○ ○ ○ ○ ○

4 ○ ○ ○ ○ ○

5 ○ ○ ○ ○ ○

>>> add\_piece(board, "X", (1,2))

True

>>> display\_board(board)

1 2 3 4 5

1 X X ○ ○ ○

2 ○ ○ ○ ○ ○

3 ○ ○ ○ ○ ○

4 ○ ○ ○ ○ ○

5 ○ ○ ○ ○ ○

>>> add\_piece(board, "Y", (2,1))

True

>>> display\_board(board)

1 2 3 4 5

1 X X ○ ○ ○

2 Y ○ ○ ○ ○

3 ○ ○ ○ ○ ○

4 ○ ○ ○ ○ ○

5 ○ ○ ○ ○ ○

>>> add\_piece(board, "Z", (1,1))

That position is not valid!

False

>>> display\_board(board)

1 2 3 4 5

1 X X ○ ○ ○

2 Y ○ ○ ○ ○

3 ○ ○ ○ ○ ○

4 ○ ○ ○ ○ ○

5 ○ ○ ○ ○ ○

#### *Task 5 move\_piece(board: list[list[str]], piece: str, current\_pos: tuple[int, int], target\_pos: tuple[int, int]) -> bool*

Moves the specified piece from current\_pos to target\_pos within board.

Positions are given as (row, column) and are 1-Indexed (That is, the top left corner is given by position (1,1)). If current\_pos does not contain piece, target\_pos is not empty, or target\_pos is not adjacent to current\_pos, then no piece is moved and INVALID\_MOVEMENT\_MESSAGE (given in support.py) should be printed to the screen. This function should return a boolean that is True if (and only if) a piece was successfully moved on the board; otherwise, it should return False.

**Tip**

This function does not return a board, instead it mutates the given board state.

Preconditions:

* current\_pos and target\_pos correspond to a position that exists within board

Example:

>>> board = create\_empty\_board(5)>>> add\_piece(board, "X", (1,1))

True

>>> add\_piece(board, "Y", (2,2))

True

>>> add\_piece(board, "Z", (3,3))

True

>>> display\_board(board)

1 2 3 4 5

1 X ○ ○ ○ ○

2 ○ Y ○ ○ ○

3 ○ ○ Z ○ ○

4 ○ ○ ○ ○ ○

5 ○ ○ ○ ○ ○

>>> move\_piece(board, "X", (1,1),(1,2))

True

>>> display\_board(board)

1 2 3 4 5

1 ○ X ○ ○ ○

2 ○ Y ○ ○ ○

3 ○ ○ Z ○ ○

4 ○ ○ ○ ○ ○

5 ○ ○ ○ ○ ○

>>> move\_piece(board, "X", (1,1),(2,1))

That movement is not valid!

False

>>> display\_board(board)

1 2 3 4 5

1 ○ X ○ ○ ○

2 ○ Y ○ ○ ○

3 ○ ○ Z ○ ○

4 ○ ○ ○ ○ ○

5 ○ ○ ○ ○ ○

>>> move\_piece(board, "X", (1,2),(2,2))

That movement is not valid!

False

>>> display\_board(board)

1 2 3 4 5

1 ○ X ○ ○ ○

2 ○ Y ○ ○ ○

3 ○ ○ Z ○ ○

4 ○ ○ ○ ○ ○

5 ○ ○ ○ ○ ○

>>> move\_piece(board, "X", (2,2),(1,1))

That movement is not valid!

False

>>> display\_board(board)

1 2 3 4 5

1 ○ X ○ ○ ○

2 ○ Y ○ ○ ○

3 ○ ○ Z ○ ○

4 ○ ○ ○ ○ ○

5 ○ ○ ○ ○ ○

>>> move\_piece(board, "Y", (2,2),(1,1))

True

>>> display\_board(board)

1 2 3 4 5

1 Y X ○ ○ ○

2 ○ ○ ○ ○ ○

3 ○ ○ Z ○ ○

4 ○ ○ ○ ○ ○

5 ○ ○ ○ ○ ○

>>> move\_piece(board, "Y", (1,1),(5,5))

That movement is not valid!

False

>>> display\_board(board)

1 2 3 4 5

1 Y X ○ ○ ○

2 ○ ○ ○ ○ ○

3 ○ ○ Z ○ ○

4 ○ ○ ○ ○ ○

5 ○ ○ ○ ○ ○

#### *Task 6 check\_input(command: str) -> bool*

Returns a boolean that is True if (and only if) the given command is of a valid format. Possible valid command formats are given in [Table 1](https://csse7030.github.io/2025s2/a1" \l "tbl-actions). Commands should be case insensitive.

Example:

>>> check\_input("p11")

True

>>> check\_input("Amogus")

False

>>> check\_input("P55")

True

>>> check\_input("A33")

False

>>> check\_input("m33")

False

>>> check\_input("m3311")

True

>>> check\_input("m3611")

False

>>> check\_input("m3301")

False

>>> check\_input("")

False

>>> check\_input("h")

True

>>> check\_input("Q")

True

>>> check\_input("hQ")

False

>>> check\_input("h ")

False

>>> check\_input(" h")

False

#### *Task 7 get\_command() -> str*

Repeatedly prompts the user until they enter a valid command. Returns the first valid command entered by the user.If the given command is not valid, INVALID\_FORMAT\_MESSAGE (given in support.py) will be printed to the screen. The possible valid commands are given in [Table 1](https://csse7030.github.io/2025s2/a1" \l "tbl-actions). valid\_moves contains the moves that the current user may make on the board. A move should not be returned if it is not in this list (the Q and H commands are not moves).

The user’s command will be case insensitive, but the returned command should be lower case.

Example:

>>> get\_command()

Please enter your command (h to see valid command): A

Invalid command. Enter 'h' for valid command format or 'q' to quit

Please enter your command (h to see valid command): H

'h'

>>> get\_command()

Please enter your command (h to see valid command): m1111

'm1111'

>>> get\_command()

Please enter your command (h to see valid command): P10

Invalid command. Enter 'h' for valid command format or 'q' to quit

Please enter your command (h to see valid command): P61

Invalid command. Enter 'h' for valid command format or 'q' to quit

Please enter your command (h to see valid command): P333

Invalid command. Enter 'h' for valid command format or 'q' to quit

Please enter your command (h to see valid command): P33

'p33'

#### *Task 8 has\_unbroken\_line(board: list[list[str]], piece: str) -> bool*

Returns True if (and only if) the given board contains an unbroken line of the specified piece at least 4 long; otherwise returns False. Lines can be horizontal, vertical, or diagonal.

Example:

>>> board = [["X","Y"],["X","Y"],["X","Y"],["X","Y"]]>>> display\_board(board)

1 2

1 X Y

2 X Y

3 X Y

4 X Y

>>> has\_unbroken\_line(board,"X")

True

>>> has\_unbroken\_line(board,"Y")

True

>>> has\_unbroken\_line(board,"Z")

False

>>> board = [["X","X","Y","Y"],["X","X","Y","Y"],["Y","Y","Y","Y"],["Y","Y","Y","Y"]]>>> display\_board(board)

1 2 3 4

1 X X Y Y

2 X X Y Y

3 Y Y Y Y

4 Y Y Y Y

>>> has\_unbroken\_line(board,"X")

False

>>> board = [["Y","Y","Y","Y"],["X","X","X","X"],["Y","Y","Y","Y"],["Y","Y","Y","Y"]]>>> display\_board(board)

1 2 3 4

1 Y Y Y Y

2 X X X X

3 Y Y Y Y

4 Y Y Y Y

>>> has\_unbroken\_line(board,"X")

True

>>> board = [["Y","Y","X","Y"],["Y","Y","X","Y"],["Y","Y","X","Y"],["Y","Y","X","Y"]]>>> display\_board(board)

1 2 3 4

1 Y Y X Y

2 Y Y X Y

3 Y Y X Y

4 Y Y X Y

>>> has\_unbroken\_line(board,"X")

True

>>> board = [["X","Y","Y","Y","Y"],["Y","X","Y","Y","Y"],["Y","Y","X","Y","Y"],["Y","Y","Y","Y","Y"],["Y","Y","Y","Y","X"]]>>> display\_board(board)

1 2 3 4 5

1 X Y Y Y Y

2 Y X Y Y Y

3 Y Y X Y Y

4 Y Y Y Y Y

5 Y Y Y Y X

>>> has\_unbroken\_line(board,"X")

False

>>> board = [["X","Y","Y","Y","Y"],["Y","X","Y","Y","Y"],["Y","Y","X","Y","Y"],["Y","Y","Y","Z","Y"],["Y","Y","Y","Y","X"]]>>> display\_board(board)

1 2 3 4 5

1 X Y Y Y Y

2 Y X Y Y Y

3 Y Y X Y Y

4 Y Y Y Z Y

5 Y Y Y Y X

>>> has\_unbroken\_line(board,"X")

False

>>> board = [["X","Y","Y","Y","Y"],["Y","X","Y","Y","Y"],["Y","Y","X","Y","Y"],["Y","Y","Y","X","Y"],["Y","Y","Y","Y","X"]]>>> display\_board(board)

1 2 3 4 5

1 X Y Y Y Y

2 Y X Y Y Y

3 Y Y X Y Y

4 Y Y Y X Y

5 Y Y Y Y X

>>> has\_unbroken\_line(board,"X")

True

#### *Task 9 has\_square(board: list[list[str]], piece: str) -> bool*

Returns true if (and only if) the given board contains a complete 2x2 square of the specified piece.

Example:

>>> board = [["X","X","Y","Y"],["X","X","Y","Y"],["Y","Y","Y","Y"],["Y","Y","Y","Y"]]>>> display\_board(board)

1 2 3 4

1 X X Y Y

2 X X Y Y

3 Y Y Y Y

4 Y Y Y Y

>>> has\_square(board,"X")

True

>>> board = [["Y","Y","Y","Y"],["X","X","X","X"],["Y","Y","Y","Y"],["Y","Y","Y","Y"]]>>> display\_board(board)

1 2 3 4

1 Y Y Y Y

2 X X X X

3 Y Y Y Y

4 Y Y Y Y

>>> has\_square(board,"X")

False

>>> board = [["Y","Y","Y","Y"],["X","X","X","X"],["Y","Y","Y","Y"],["Y","Y","Y","Y"]]>>> display\_board(board)

1 2 3 4

1 Y Y Y Y

2 X X X X

3 Y Y Y Y

4 Y Y Y Y

>>> has\_square(board,"Y")

True

>>> board = [["Y","Y","Y","Y"],["X","X","X","X"],["Y","Y","Y","Y"],["Y","Y","Y","Y"]]>>> display\_board(board)

1 2 3 4

1 Y Y Y Y

2 X X X X

3 Y Y Y Y

4 Y Y Y Y

>>> has\_square(board,"Z")

False

>>> board = [["X","X"],["X","X"]]>>> display\_board(board)

1 2

1 X X

2 X X

>>> has\_square(board,"X")

True

#### *Task 10 check\_win(board: list[list[str]]) -> str*

Returns the piece of the player who has won within the given board state according to the game rules specified in [the gameplay section](https://csse7030.github.io/2025s2/a1" \l "gameplay). If no player has won, then an empty space should be returned. If both players have won, then player one takes precedence (and should be returned). Only player one and player two can win.

P1 = PLAYER\_1\_PIECEP1

'◍'

P2 = PLAYER\_2\_PIECEP2

'●'

board = create\_empty\_board(5)display\_board(board)

1 2 3 4 5

1 ○ ○ ○ ○ ○

2 ○ ○ ○ ○ ○

3 ○ ○ ○ ○ ○

4 ○ ○ ○ ○ ○

5 ○ ○ ○ ○ ○

check\_win(board)

'○'

add\_piece(board, P1, (1,1))

True

add\_piece(board, P1, (2,2))

True

add\_piece(board, P1, (3,3))

True

add\_piece(board, P2, (1,4))

True

add\_piece(board, P2, (1,5))

True

add\_piece(board, P2, (2,4))

True

display\_board(board)

1 2 3 4 5

1 ◍ ○ ○ ● ●

2 ○ ◍ ○ ● ○

3 ○ ○ ◍ ○ ○

4 ○ ○ ○ ○ ○

5 ○ ○ ○ ○ ○

check\_win(board)

'○'

add\_piece(board, P2, (2,5))

True

display\_board(board)

1 2 3 4 5

1 ◍ ○ ○ ● ●

2 ○ ◍ ○ ● ●

3 ○ ○ ◍ ○ ○

4 ○ ○ ○ ○ ○

5 ○ ○ ○ ○ ○

check\_win(board)

'●'

add\_piece(board, P1, (4,4))

True

display\_board(board)

1 2 3 4 5

1 ◍ ○ ○ ● ●

2 ○ ◍ ○ ● ●

3 ○ ○ ◍ ○ ○

4 ○ ○ ○ ◍ ○

5 ○ ○ ○ ○ ○

check\_win(board)

'◍'

board = [["X","X"],["X","X"]]display\_board(board)

1 2

1 X X

2 X X

check\_win(board)

'○'

#### *Task 11 play\_game() -> None*

Coordinates gameplay of a single game of Teeko from start to finish according to the [Gameplay](https://csse7030.github.io/2025s2/a1" \l "gameplay) section (excluding steps 7 and 8). The print output from your play\_game function (including prompts) must exactly match the expected output. In order to help you match output exactly, most messages that should be printed during gameplay are provided as constants in support.py. 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.

**Tip**

turn\_message has been provided in support.py to help get you started. You may or may not choose to use it.

#### *Task 12 main() -> None*

The main function should be called when the file is run. The main function handles steps 7 and 8 from the [Gameplay](https://csse7030.github.io/2025s2/a1" \l "gameplay) section. It repeatedly enacts a game of Teeko using the play\_game function until the user specifies they no longer wish to continue. AGAIN\_PROMPT (given in support.py) provides the prompt you should use when asking the user for input.

The gameplay/ folder provided with this assignment contains 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.

## Assessment and Marking Criteria

This assignment assesses following 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.

There are a total of 100 marks for this assessment item.

### Functionality

Your program’s functionality will be marked out of a total of 60 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 for partially working functions, or for implementing only a few functions.

**Warning**

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.

### 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 40.

You are expected to follow the PEP-8 style guidelines discussed in lectures. 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, and ensuring all lines are below 80 characters.
  + 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. Comments must be written in English.
  + 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 (in English) 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.

### 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 CSSE7030. You will see a list of assignments. Choose Assignment 1. You will be prompted to choose a file to upload. Do not select the option to submit using a git repository if it is presented to you. 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](https://edstem.org/au/courses/24510) 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 [myUQ](https://my.uq.edu.au/). You must retain the original documentation for a minimum period of six months to provide as verification, should you be requested to do so.

### Plagiarism and Referencing AI Use

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.

Artificial Intelligence (AI) and Machine Translation (MT) are emerging tools that may support you in completing this assessment task. You may appropriately use AI and/or MT in completing your assessment tasks. However, you ****must clearly reference**** any use of AI or MT in each instance. A failure to reference generative AI or MT use may constitute student misconduct under the Student Code of Conduct. Please refer to [AI Guideline](https://learn.uq.edu.au/ultra/courses/_192509_1/outline/edit/document/_11738816_1?courseId=_192509_1&view=content&state=view) that has been provided on the Ultra Blackboard page of the course.

**Important**

You ****must**** understand all code submitted and submit a ****PDF file**** alongside with your final code submission. This file must provide detailed information about how the AI tools were used. Please refer to [AI Guideline](https://learn.uq.edu.au/ultra/courses/_192509_1/outline/edit/document/_11738816_1?courseId=_192509_1&view=content&state=view) for guidance on properly documenting interactions with AI.

You are encouraged to complete both parts A and B of the [academic integrity modules](https://web.library.uq.edu.au/library-services/it/learnuq-blackboard-help/academic-integrity-modules) before starting this assignment. Submitted assignments will be electronically checked for potential cases of plagiarism.