COURSEWORK SPECIFICATION
ECM1400 – Programming
Module Leader: Abhijit Chatterjee
Academic Year: 2024/25

Title: **Continuous Assessment 2: Battleships**

Submission deadline: **Mon, 9 Dec 2024, 12:00**

This assessment contributes **70%** of the total module mark and assesses the following **intended learning outcomes**:

- 1 design an algorithm, using sequence, iteration and selection
- 2 write, compile, test, and debug a computer program
- 4 systematically test your programs
- 5 document software to accepted standards
- 9 systematically break down a problem into its components
- 10 understand and choose appropriate programming techniques
- 11 analyse a problem and synthesise a solution
- 12 use technical manuals and books to interpret specifications and technical errors

This is an individual assessment and you are reminded of the University's regulations on collaboration and plagiarism. You must avoid plagiarism, collusion, and any academic misconduct behaviours. Further details about academic honesty and plagiarism can be found at https://ele.exeter.ac.uk/course/view.php?id=1957.

The Use of Generative AI

Use of GenAI tools in Continuous Assessment 2 for ECM1400 – Programming.

The University of Exeter is committed to the ethical and responsible use of Generative AI (GenAI) tools in teaching and learning, in line with our academic integrity policies where the direct copying of AI-generated content is included under plagiarism, misrepresentation and contract cheating under definitions and offences in TQA Manual Chapter 12.3. To support students in their use of GenAI tools as part of their assessments, we have developed a category tool that enables staff to identify where use of Gen AI is integrated, supported or prohibited in each assessment. This assessment falls under the category of AI Prohibited. This is because it is important for the program created to be your own work in order to meet the learning objectives outlined above. You can find further guidance on

using GenAI critically, and how to use GenAI to enhance your learning, on Study Zone digital.

When submitting your assessment, you must include the following declaration, ticking all that apply:

I have not used any GenAI tools in preparing this assessment.

I declare that I have referenced use of GenAI outputs within my assessment in line with the University referencing guidelines.

Please note: Submitting your work without an accompanying declaration, or one with no ticked boxes, will be considered a declaration that you have not used generative AI in preparing your work

If a declaration sheet cannot be uploaded as part of an assignment (i.e. at the start of an essay), students understand that by submitting their assessment that are confirming they have followed the assessment brief and guidelines about GenAI use.

Instructions

The Battleship game is a strategic guessing game typically played by two players. In the traditional board game, each player has a board with a grid of squares, typically 10x10, where they can place a fleet of ships without the opponent's knowledge. The objective is to sink your opponent's fleet before they sink yours. Each player takes turns to declare coordinates for their strike and the opponent replies whether that is a hit or miss until one player has no battleships remaining.

In this assignment you will build a Python application to simulate the Battleships game. The specification for how the components of the game must be built are specified below. Some features of your application are strictly defined and specific. It is important that you adhere to these instructions as they will be tested with a testing framework so must be implemented precisely else the tests will fail. Many of the later features are less precise and open to be extended. These are intentionally less precise to enable individuals to distinguish their project from other students. In order for us to understand what has been implemented in this part of the assessment any additional features must be described with bullet points in 100 words at the top the project README file.

This document will start by outlining the concept of the game for those not familiar with Battleships then go into the technical specification for the Python application.

Key aspects of the game:

1. **Setting up the Fleet:** Before the game begins, each player arranges their ships on their own grid without showing the opponent the placement. The ships vary in size: commonly, there are five ships per player. The types of ships often include:
    - Aircraft Carrier (5 spaces)
    - Battleship (4 spaces)
    - Cruiser (3 spaces)
    - Submarine (3 spaces)
    - Destroyer (2 spaces)
2. **Gameplay:** Players take turns calling out specific grid coordinates (for example, "B-5" or "H-7") to identify a location on the opponent's grid to attack. The opponent then responds with "Hit" if the coordinate is part of a ship or "Miss" if it's an empty space. Players continue taking shots alternately until one player's entire fleet is sunk.
3. **Strategy:** Success in Battleship involves a combination of luck and strategic deduction. Players try to systematically target the opponent's ships based on previous hits and

misses. Once a hit is confirmed, they use it to determine the orientation and position of the ship and concentrate fire in that area.

4. **Winning:** The game ends when one player successfully sinks the entire fleet of the opponent. The winner is the one who manages to do this first.

The game can be played with variations in rules, grid size, and ship types. The simplicity and strategic thinking required in Battleship make it a popular and classic two-player game enjoyed by many.

Battleships as a Python application

Now you understand the premise of the Battleships game, it is your turn to implement this into a Python application with a command-line interface and web interface to enable a user to play against an AI opponent.

You will start by building the basic components of the game and creating a one-player game loop with no opponent on a 10x10 board with 5 battleships to demonstrate the basic components of the game are functional.

Game components:

Create a module called ***components.py*** that contains the following functions to setup the components of the game:

- Initialise board

Create a function called ***initialise_board*** with a single argument, *size*, that has a default value 10 and returns a list of lists that represents the board state. The initial board state should be empty so the list should be full of None values.

- Create battleships

Create a function called ***create_battleships*** that takes one optional argument, *filename*, which has a default value "battleships.txt". The function should read the file and create a dictionary variable with the identifier, *battleships,* containing keys as the battleship name and the value as its size. This dictionary should be returned. No representation of position of each battleship is needed at this stage, only the name and size.

An example data structure that would be returned is provided below. This is a dictionary with two battleships, 'Ship1' and 'Ship2', with sizes of 3 and 2 respectively.

```
{'Ship1': 3, 'Ship2': 2}
```

- Place battleships

Create a function called ***place_battleships*** with two arguments, *board* and *ships*.
The *board* argument should be a list of lists that was returned from an *initialise_board* function call, and, *ships* should be a dictionary that was returned from a *create_battleships* function call. The place_battleships function should then update the board data structure to position the ships on the board. The function should return a list of lists representation of the board with the ship name in the value in each list item where the battleship is placed. An example board representation for a 3x3 board has been provided below to demonstrate the expected data structure if two ships from the previous demonstration were placed on the first and last row.

```
[['Ship1','Ship1','Ship1'],

[None,None,None],

['None,'Ship2','Ship2']]
```

There are multiple algorithms possible to place the ships on the board. As an extension of the core functionality **add an optional algorithm argument** with default value of 'simple' that can be extended to include more sophisticated algorithms for placing ships.

Simple - The first implementation should place each battleship horizontal on a new rows starting from (0,0).

Random - A second implementations should place the battleships in random positions and orientations.

Custom – A third implementation should allow the battleships to be placed according to a placement configuration file, placement.json.

The placement algorithm may be extended in a later feature to create difficulty levels.

Now you have the components of the game you will need to manage the game mechanics. Create a module called ***game_engine.py*** with the following functions:

- Attack

Create a function called ***attack*** that takes three arguments, a tuple called *coordinates*, e.g. (1, 1), a *board* and *battleships*. Implement the function to check is there is a battleship at that coordinate on the board for the corresponding attack. The function should return True if there is and this is a 'Hit' else if it is a 'Miss' the function should return False. Additionally, the coordinates position in the board data structure should be updated to *None* and the size value in the *battleships* dictionary should be decremented. If the value in the *battleships* data structure is decremented to zero, you have sunk that battleship.

- Command-line input for attack coordinates

Create a function called ***cli_coordinates_input*** that takes no arguments. In this function request the user to input coordinates for an attack and process the input to return a tuple containing x and y coordinates, e.g. *(1, 4)*.

- Simple game loop

Create a function called ***simple_game_loop*** with no arguments. This is for intermediate manual testing through the command-line interface. Within the *simple_game_loop* you should implement the following steps.

1. Start the game with a welcome message.

2. Initialise the board, the ships and place the ships using the default settings to create the game components.

3. Prompt the user to input coordinates of their attack.

4. Process the attack on the single board created above. Print a hit or miss message to the user each time.

5. Repeat steps 4 and 5 until all battleships have been sunken.

6. Print game over message.

Call the *simple_game_loop* function from a ***if __name__ == __main__*** construct in the global namespace of the ***game_engine.py*** module so *simple_game_loop* can be manually tested by executing the module.

Well done, at this stage you have a single player game. However, games aren't much fun on your own as you know the solution. Next you will use your create the logic for multiple players, including an AI opponent and add a user interface.

Multi-player constructs

- Players

Create a module called ***mp_game_engine*** and create a dictionary variable in the global namespace called *players*. Store the username of each player as the key and their board and battleships as the value.

- AI opponent

Create a function called ***generate_attack*** that returns a tuple with coordinates that can be passed to the *attack* function. The coordinate produced by this function should change each time the function is called and should always be within the size of the board in play.

- AI opponent command-line game loop

Create a function called ***ai_opponent_game_loop*** with the following steps to allow game play through the command-line interface:

1. Start the game with a welcome message.
2. Initialise two players in the dictionary, create their battleships and board. One player will be the user and the second player will be the AI opponent. Place the battleships using the random placement algorithm for the AI opponent and custom placement algorithm for the user. Remember, there should be a placement.json file that can be updated by the user player to configure their setup.
3. Prompt the user to take their turn and input the coordinates of their attack.
4. Process the attack on the board of the AI opponent player. Print a Hit or Miss message to the user each time.
5. Generate an attack from the AI opponent against the user player and process the attack on the board of the user player. Print a Hit or Miss message to the user each time. When reporting the AI opponent Hit or Miss, include an ascii representation of the user player's board after each turn.
6. Repeat steps 3 to 5 until all battleships for one player have been sunken.
7. Print appropriate game over message depending on whether the user won or lost.

Well done. At this stage you have a fun command-line game of battleships with a simple AI opponent. However, it's the interface is basic in the command-line and the coordinate input and battleship placement aren't intuitive. Next we're going to add a graphical user interface.

The graphical user interface using Flask

Create a new module called ***main.py*** which will become the main entry point to our project. In this module you should create two functions to handle the two webpage interfaces. In the global namespace you should also create an app variable from the Flask object and initiate the application from the if __name__=='__main__' block using app.run() to launch the web server that will handle the interface. If this is not familiar don't worry instructions and examples on how to use Flask will be available in the workshops in week 8 and 9. There will be two main webpages for our interface, the Battleships Placement page, and, the Gameplay page.

- Battleship Placement page

Create a function called ***placement_interface*** with a Flask decorator to map the /placement url endpoint. In this function you must return a call to the Flask function render_template and use the placement.html page template provided in the ELE2 assessment tile.

- Gameplay page

Create a function called ***root*** with a Flask decorator to map the empty "/" url endpoint. This function should be mapped to the main gameplay template webpage. In this function you will need to handle request variables to detect if the web request has get or post variables. If the request contains the variables to setup the board a new game has been initiated. If the request

contains a coordinate, the user has input their selection and the game logic should be progressed.

Now you have the user interface constructs, you will need to integrate the logic from the command-line game with the user interface. The instructions in this step are less precise and individual solutions are expected to diverge at this stage so will be tested manually.

Congratulations, if you've completed all the steps to here you've implemented a web based Battleships game with an AI opponent and completed the major remit of the assessment. The final requirements are designed to stretch the most capable students and will require much more time to achieve fewer marks.

Difficulty levels

In order to layer further complexity into the game there is an opportunity to extend the standardised game framework provided. In order to extend the functionality add a set of configuration options that enable the game to include difficulty levels. To achieve this you may extend the following features:

Improve the AI opponent:

- Add a more sophisticated Battleship placement algorithm to add difficulty levels. This can be done through a range of pre-calculated optimal placement configuration or an adaptable model based algorithm. Of course, pre-calculated placements don't directly scale to changing board size.
- Improve the ***generate_attack*** function to create a better AI opponent. The accuracy can be significantly improved if there is a registered hit but the ship is not sunken.
- Make the game dynamics change over time. The AI opponent could learn from previous positions by creating a history. If you are feeling really ambitious you may also consider allowing the ships to move in a given direction by one space on each turn.

Add multi-player interface to the webpage:

- The client server architecture of a web interface could be ideal for multi-player gameplay with peers on remote devices. However, you need to consider how to synchronise two player mode where two users can receive updates from each others turn.
- When playing it can be useful to know how many ships you have left on your own board. Update the the gameplay webpage template and function so that the user can view their own board state alongside the previous input they have sent to attack their opponent.

Submission

Code will be submitted as a zip file in ELE2. All source code required to run the project must be submitted and all documentation and remote repositories containing the code should be linked and clearly referenced.

The README file should include a list of all features that you consider are complete followed by a typical README file structure. The self-assessment must be less than 200 words and understandable within 1 minute. The subsequent README must make it clear how to execute the project, the license and any other relevant details.