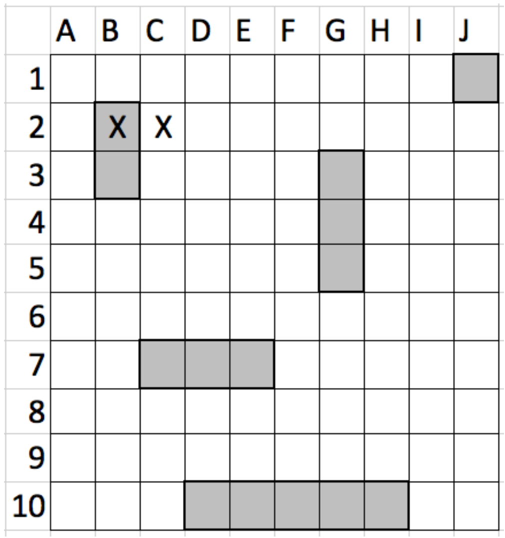
Coursework 3: Battleship Game

COMP70053 Python Programming (Autumn 2021)

Deadline: Monday 15th Nov 2021 (7pm GMT) on CATe



1 Overview

In this coursework, you will implement a Battleship game.

The objectives of this coursework are:

- to enable you to apply everything you have learnt in the core lessons to tackle a larger programming problem;
- to allow you to practise applying object-oriented concepts like composition/aggregation and inheritance;
- to give you hands-on experience with object-oriented programming in a practical (and hopefully fun) application.

This coursework covers topics in the Core Lessons 1 to 10 of the course materials.

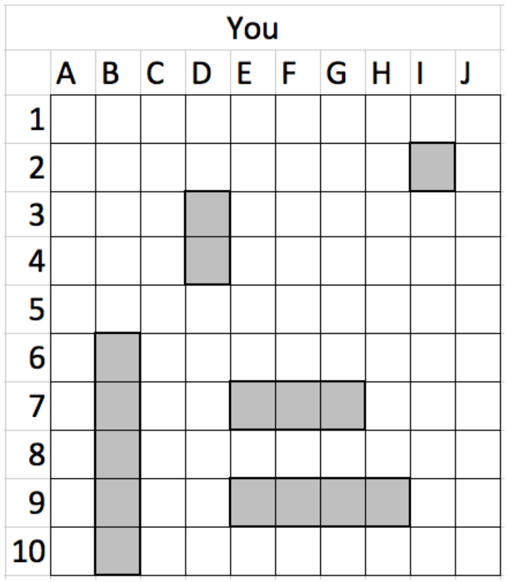
You are limited to the **Python Standard Library** for this coursework. Do **NOT** use any external libraries such as `NumPy`.

2 Description of the Battleship Game

Battleship is a strategy type guessing game for two players. Players mark their ships on an $N \times N$ grid. Players then alternate turns to hit the other player's ships. The objective of the game is to destroy the opposing player's fleet. In this coursework we look at a variant of the Battleship game.

Before the game starts, players arrange some ships on their board. A ship is of size $1 \times L$, where L is the number of consecutive squares it occupies on the grid. L can be 1 to 5 (inclusive). A ship can be arranged vertically ($L \times 1$) or horizontally ($1 \times L$). The ships also cannot overlap, i.e. only one ship can occupy any given square in the grid.

Players are provided with two boards: one on which players arrange their ships and record the opponent's attacks towards their ships; and another 'tracking' board on which players record their own attacks towards their opponent's ships (Figure 1).



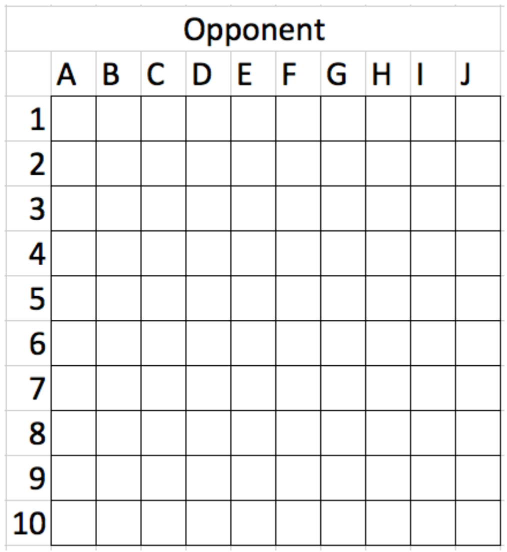


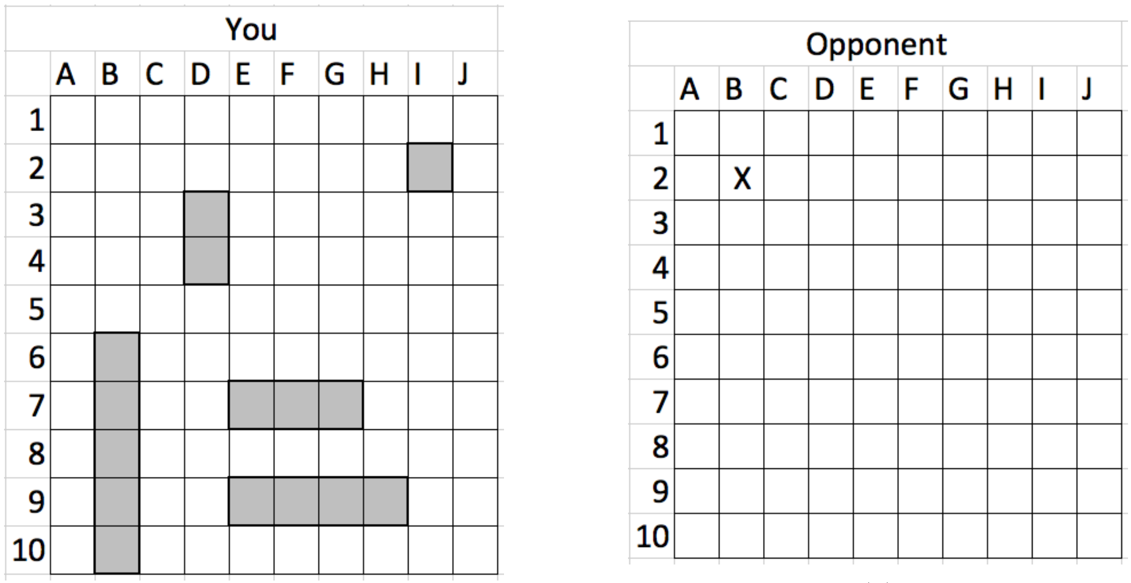
Figure 1: The board on the left contains your ships and where you record your opponent's attacks towards your ships. The 'tracking' board on the right is the one on which you keep track of your attacks towards your opponent's ships.

The game then starts, where players take turns as follows:

- one player announces a target cell in the opponent's grid at which to attack
- the opponent announces whether or not a ship occupies the cell
  - if occupied (i.e. the ship is 'hit'), the victim marks the cell on his board with an **X**
  - when all cells of a ship have been hit, the victim announces 'sink'
  - the attacker marks the hit/miss on his/her own 'tracking' board
  - the attacker then launches another attack until he/she misses.

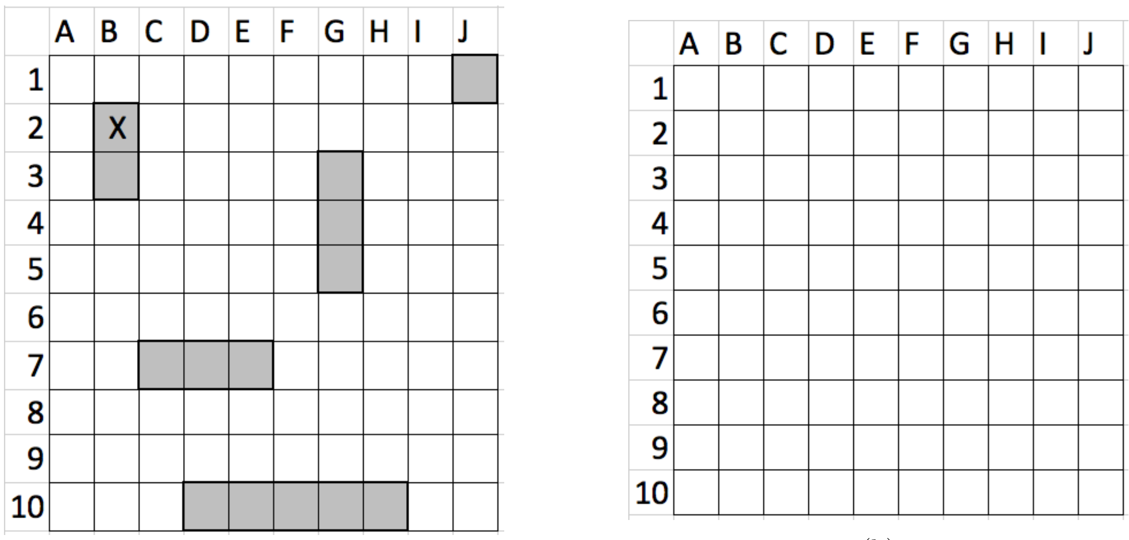
After a miss, it will be the opponent's turn to attack, and the process is repeated. A player wins the game when all of the other player's ships have been sunk.

As an example, let's say you are the current attacker. You pick a target cell, say **B2**. You record the shot at B2 on the tracking board (Figure 2b). The opponent says the cell is occupied by a ship and marks the attack on his own board (Figure 3a). As you have hit a cell occupied by a ship, you can have another go.



(a) (b)

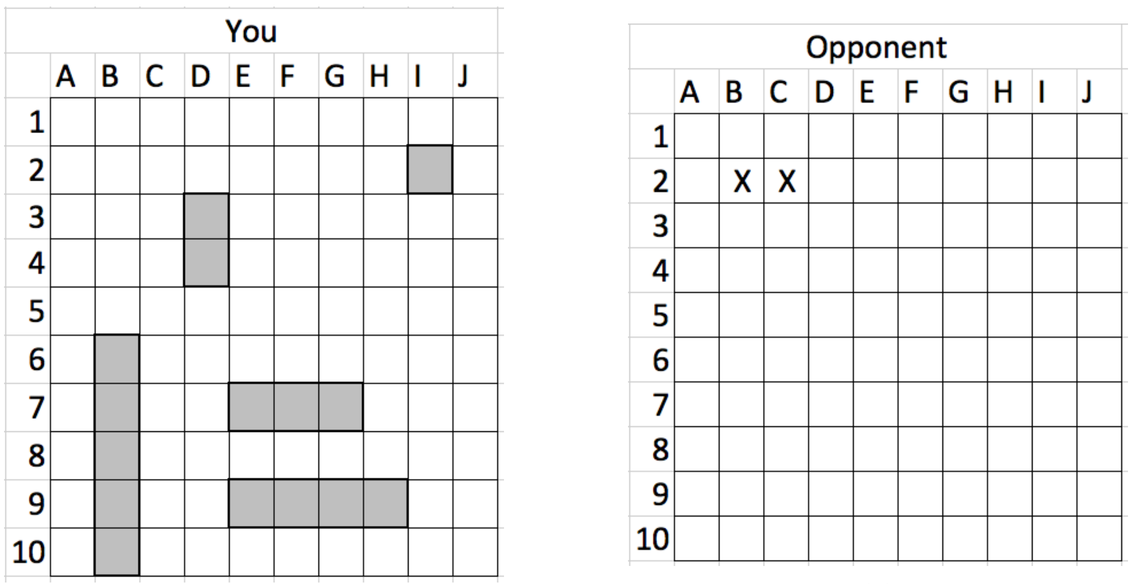
Figure 2: Your boards after the first round.



(a) (b)

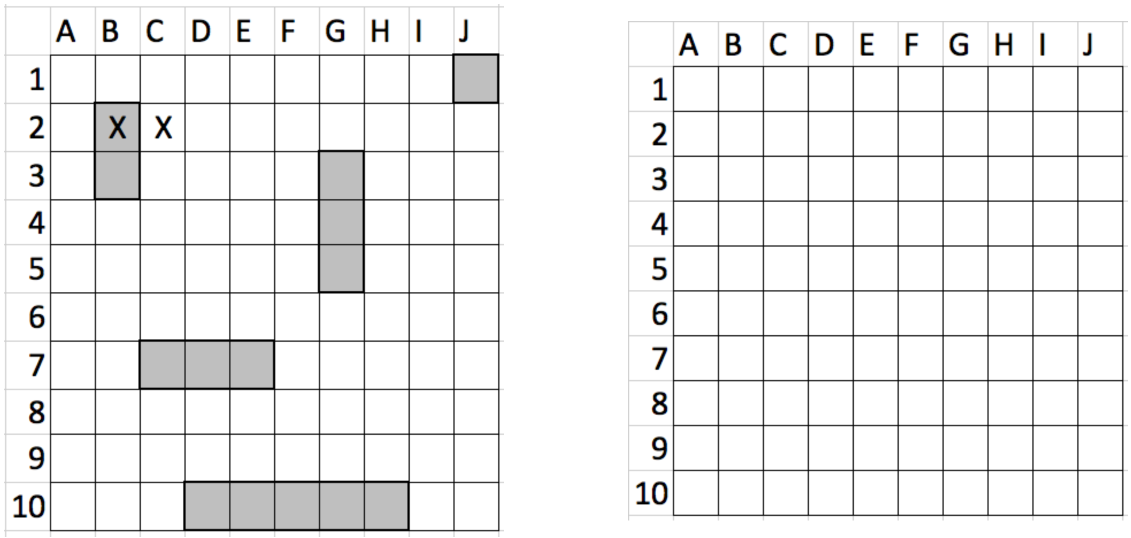
Figure 3: The opponent's boards after the first round.

Next, say you picked cell **C2**. You record the shot at C2 on your tracking board (Figure 4b). The opponent says the cell is not occupied by a ship and marks the attack on his own board (Figure 5a). Now it is the opponent's turn to attack.



(a) (b)

Figure 4: Your boards after the second round.



(a) (b)

Figure 5: The opponent's boards after the second round.

The opponent attacks at **E5**. You record the shot at E5 on your board (Figure 6a) which is not occupied by a ship and the opponent marks the shot on his/her own tracking board (Figure 7b). The game continues until either one of you have all your ships sunk by the other.

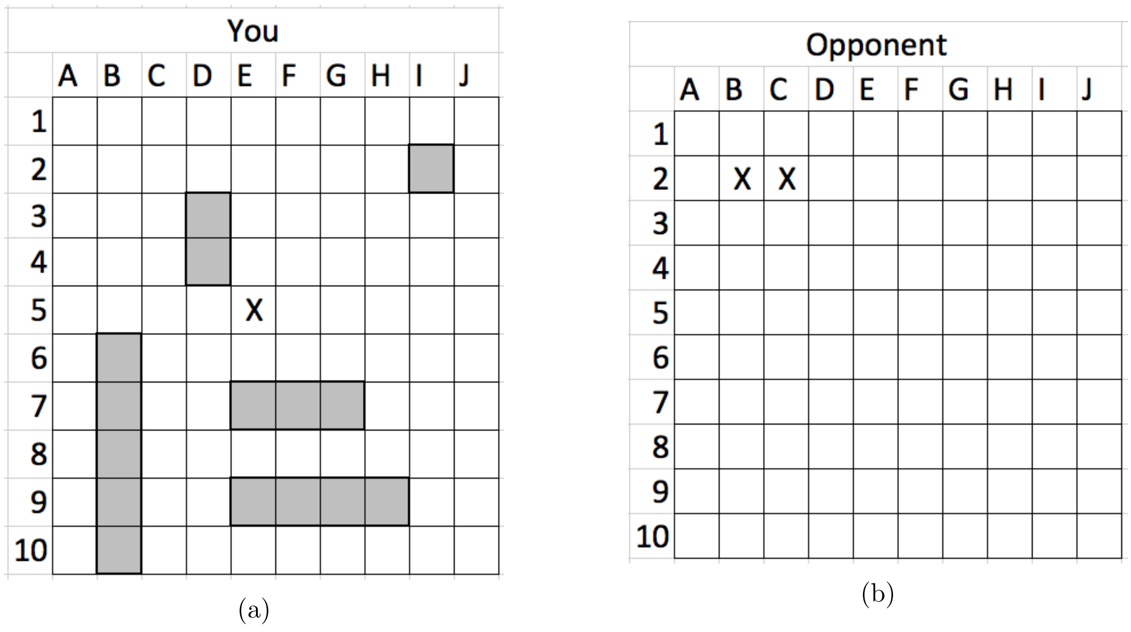


Figure 6: Your boards after the third round.

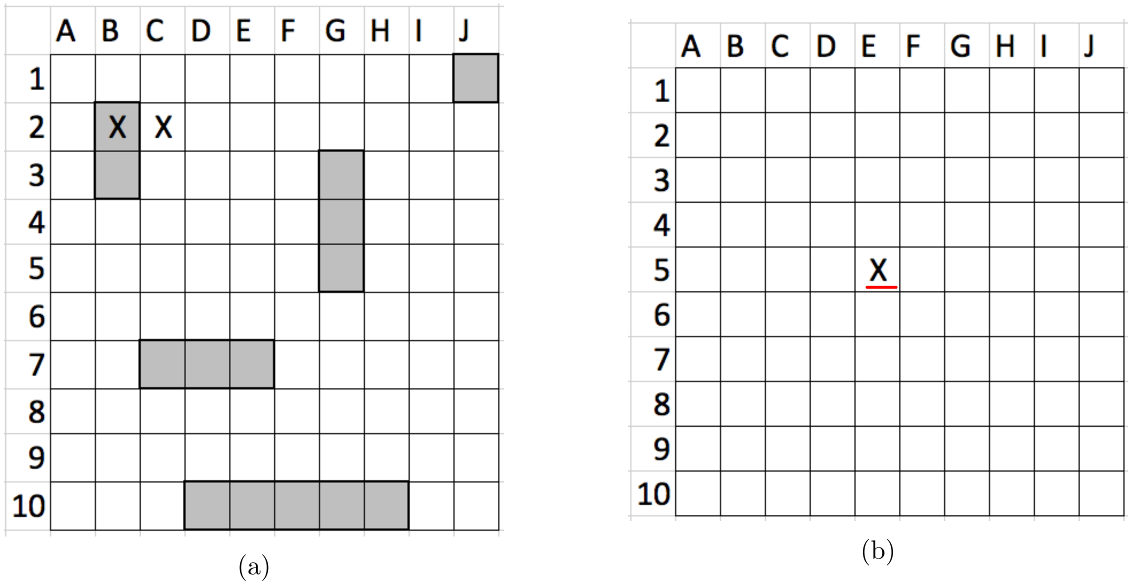


Figure 7: The opponent's boards after the third round.

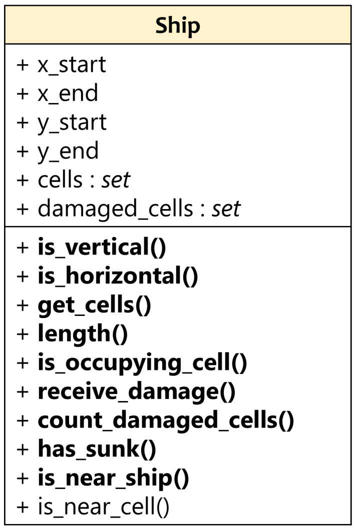
3 Tasks

You are supplied with some skeleton code (see section 4).

If you have read the description of the game, you would have identified two key objects in the game: `Ship` and `Board`. You will start by implementing these two classes, before moving on to enhance the game with some clever AI!

3.1 Task 1: Implement the `Ship` class

Your first task is to complete the `Ship` class in `battleship/ship.py`. Examine the class diagram below.



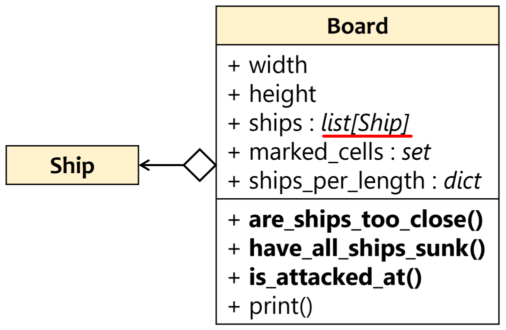
Your task is to **complete all methods listed in bold** above. Please read the docstrings in `battleship/ship.py` carefully for a detailed description of the requirements for each method.

Hints:

- This class is self-contained and does not depend on any other class.
- Most methods can be completed with minimal code (some even in one or two lines).
- Try completing methods in the given order.
- Note that the **coordinate values start from 1**. So the top-left grid cell is $(1, 1)$.
- Some example usage is provided the end of `battleship/ship.py`. Run them with `python3 -m battleship.ship`. Feel free to edit.
- You may also write your tests in `tests/test_ship.py` and run the tests with `python3 -m tests.test_ship`.
- We will not be assessing your test cases (so no marks will be awarded for these).

3.2 Task 2: Implement the `Board` class

Your next task is to complete the Board class in `battleship/board.py`. Examine the class diagram below.



Again, your task is to **complete all methods listed in bold** above. Read the docstrings carefully for details of the methods.

Hints:

- A `Board` contains a list of `Ship` instances. Naturally, you should make full use of what you have developed in Task 1 to complete this task. Every method will involve the ships!
- You might need more code per line than Task 1, but it is still possible to complete each in 10 lines of code or less. Do not worry if you end up with longer pieces of codes, as long as it behaves correctly!
- Feel free to add your own methods where necessary.
- Again, see the end of `battleship/board.py` for example usage, and `tests/test_board.py` for an example test. Free free to edit/add more tests. We will not assess your test cases.

Running a simulation

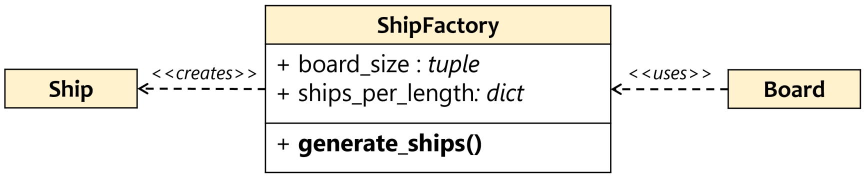
Once you have completed Task 2, you should be able to play a game between two players by running '`python3 main.py`'. Use this to check that your implementation is working correctly! If you need to cheat, you can check out the ship's locations in the `ManualVsManualSimulation` class in `battleship/simulation.py`.

3.3 Task 3: Implement the `ShipFactory` class

Now, you have a fully working framework where you can have two users play the game manually.

One weakness right now is that you will have to manually define and arrange the ships. Would it not be nice if the ships can be automatically created and arranged at random on the board?

So, your next task is to complete the `ShipFactory` class in `battleship/ship.py` to generate a list of ships automatically. See the class diagram below (only relevant methods are shown).



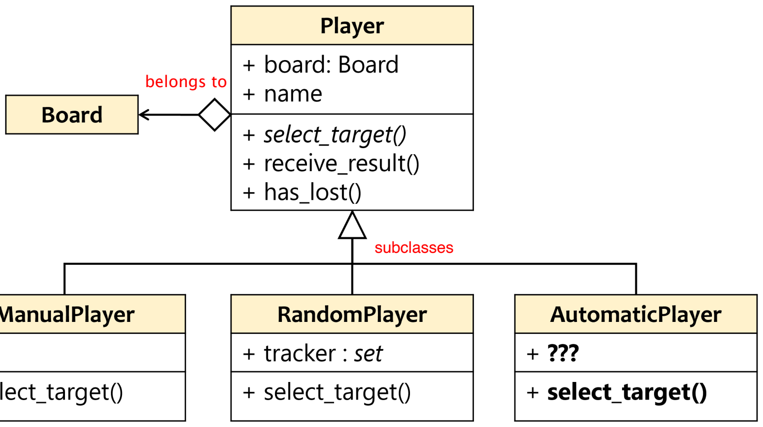
Your task is to **complete the `generate_ships()` method** to generate a list of `Ship` instances. This method will be used by `Board` to automatically generate its `Ship` instances when not specified by the user. Again, read the docstring of the class and method for a detailed description.

Notes:

- The number of generated ships for each given length must adhere to the `ships_per_length` attribute (a `dict`).
- The `Ship` instances must not be too close to each other (as defined in Task 1).
- The `Ship` instances should be within the bounds of the `board_size = (width, height)` attribute.
- The provided `create_ship_from_str()` method is not needed for the task. But please **do not modify this!**
- You can assume that any given `ships_per_length` will be reasonably similar in scale to the default. We will not give you excessively difficult cases that you will find hard to arrange!
- Feel free to add your own attributes/methods where necessary, as long as `generate_ships()` is implemented as a minimum!

3.4 Task 4: Implement the `AutomaticPlayer` class

Finally, here is arguably the most interesting task. Let us create an AI player against whom you can play! You can also pit two AI players against each other!



If you examine `battleship/player.py`, you will find a `Player` class implemented along with subclasses `ManualPlayer` and `RandomPlayer`.

`Player` has a `select_target()` method that is not implemented. This is considered an *abstract method*. It is the responsibility of its subclasses to implement this.

`ManualPlayer` implements `select_target()` by reading the cell coordinates from the user prompt. `RandomPlayer` implements it by generating random coordinates to attack. It also 'remembers' previous attacks by storing the information in its instance variables.

Your final task is to **complete the `AutomaticPlayer` class** in `battleship/player.py` by implementing a simple AI player. It should define its strategy by implementing the `select_target()` method. Your strategy should be better than random. Document your code in the class and method to clearly explain the strategy of your player.

You are free to add as many attributes and methods as necessary, as long as `select_target()` is implemented as a minimum.

Running a game

You can run multiple types of games in `main.py`. For example, running '`python3 main.py 3`' will allow you to play against your AI player, and running '`python3 main.py 5`' will pit your AI player against itself! See `main.py` for more options, and feel free to add your own simulations in `battleship/simulation.py`. Enjoy!

4 Skeleton code

You are provided with a git repository containing the skeleton code. You will use `git` to push your code changes to the repository. To obtain the code, clone the repository using either SSH or HTTPS, replacing `<login>` with your College username.

- `git clone git@gitlab.doc.ic.ac.uk:lab2122_autumn/python_cw3_<login>.git`
- `git clone https://gitlab.doc.ic.ac.uk/lab2122_autumn/python_cw3_<login>.git`

Test your code with `LabTS` (`https://teaching.doc.ic.ac.uk/labts`) after pushing your changes to GitLab. Make sure your code runs successfully on LabTS without any syntax errors. You will lose marks otherwise. These automated tests are purely to ensure that your code runs correctly on the `LabTS` servers. **You are responsible for testing your own code more extensively**. The final assessment will be performed using a more extensive set of tests not made available to you.

5 Handing in your coursework

Go to `LabTS` (`https://teaching.doc.ic.ac.uk/labts`), and click on the "`Submit to CATe`" button next to the specific commit you wish to submit. This will automatically submit your commit SHA1 hash to CATe. Double check that everything is in order on CATe.

6 Grading scheme

Criteria	**Max**	**Details**
Algorithm design and code correctness (Tasks 1+2+3+4)	9+5+5+5	Does your code run without any syntax errors? Does your code give the expected output for valid test cases, whether they be standard or edge cases? Does your program crash? Does your program not end up in an infinite loop for certain inputs? Is your ship generation strategy sound (for Task 3)? Is your AI strategy sound (for Task 4)?
Code readability	6	Did you use good coding style and naming conventions? Did you use meaningful names? Is your code highly abstracted, modular, and self-explanatory? Did you provide useful, meaningful comments where necessary? Do not hard code any 'magic numbers' like the number of ships in your code - this is bad programming practice!
TOTAL	**30**	Will be scaled to 10% of your final module grade.

Credits

Coursework designed and prepared by Luca Grillotti and Josiah Wang