BATTLESHIPS

Purposes of this assignment:

- To give you more experience with collections (tuples, sets, ...)
- To give you experience of writing tests and TDD-style development
- To give you experience of working with output and graphics

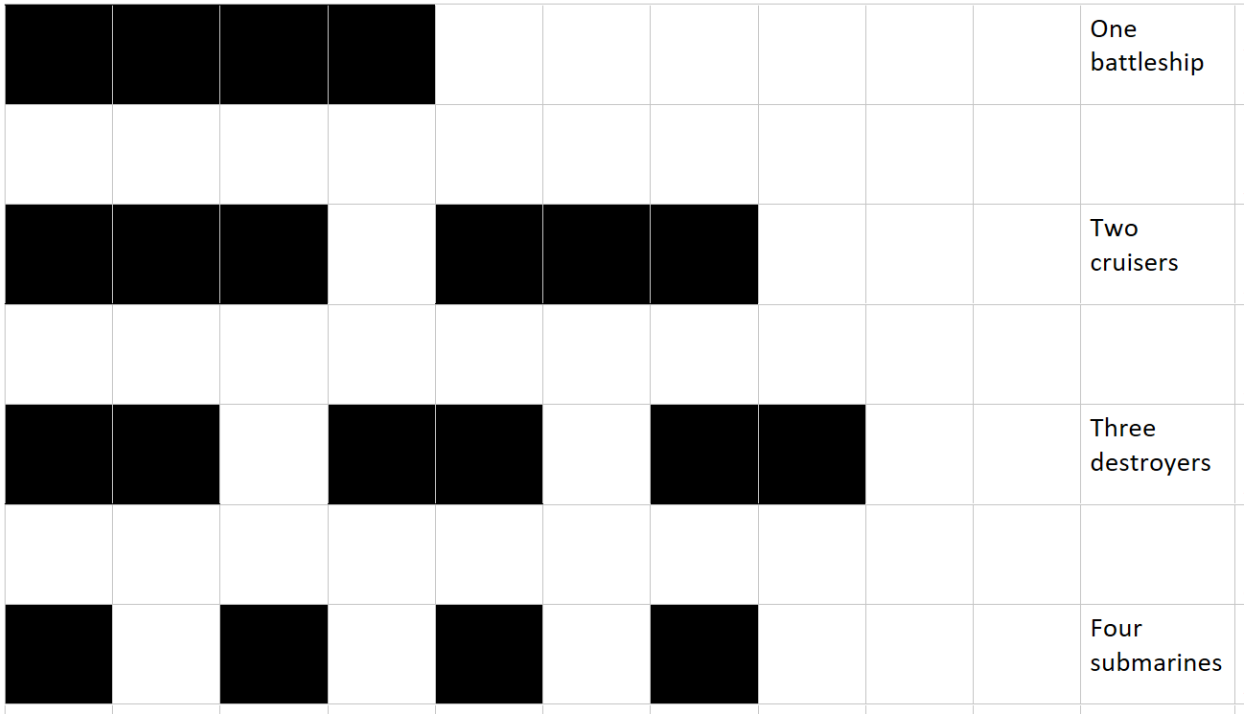
General idea of the assignment

This assignment is based on the well-known game. Battleship is usually a two-player game, where each player has a fleet and an ocean (hidden from the other player), and tries to be the first to sink the other player's fleet. We'll just do a solo version, where the computer places the ships, and the human attempts to sink them.

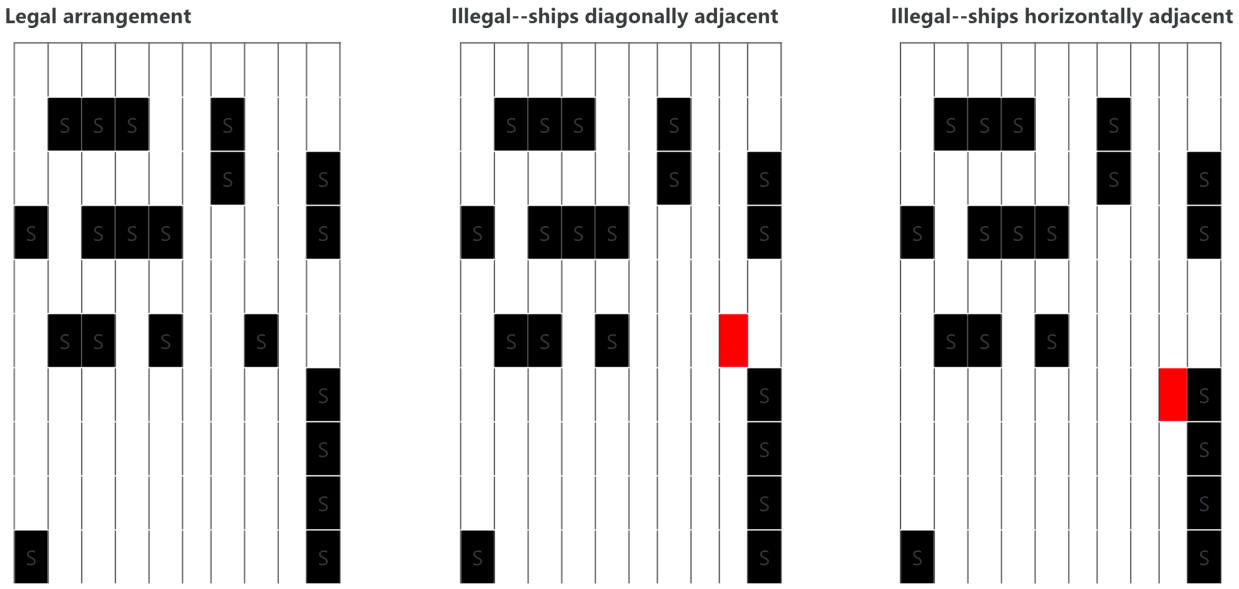
The Ocean is a field of 10 x 10 squares. The squares are numbered from 0 to 9 in eachdimension with numbers increasing from top to bottom and from left to right.

The fleet consists of 10 ships. The fleet is made up of 4 different types of ships, each of different size as follows:

- One battleship, occupying 4 squares
- Two cruisers, each occupying 3 squares
- Three destroyers, each occupying 2 squares
- Four submarines, each occupying 1 square



To begin the game, the computer places all the 10 ships of the fleet in the ocean randomly. Each ship can be placed either horizontally (as shown in the figure above) or vertically. Moreover, no ships may be immediately adjacent to each other, either horizontally, vertically, or diagonally. Examples of legal and illegal arrangements are shown below:



The human player does not know where the ships are. The human player tries to hit the ships, by calling out a row and column number. The computer responds with one bit of information--"You have a hit!" or "You missed!" (Note that the human player can call out the same location more than once, even though it does not make sense. If that happens, 2nd, 3rd, ... calls of the same location are misses.) When a ship is hit but not sunk, the program does not provide any information about what kind of a ship was hit. However, when a ship is hit *and* sinks, the program prints out a message `"You sank a _ship-type_!"`

A ship is "sunk" when every square of the ship has been hit. Thus, it takes four hits (in four different places) to sink a battleship, three to sink a cruiser, two for a destroyer, and one for a submarine. The objective is to sink the fleet with as few shots as possible; the best possible score would be 20. (Low scores are better.) When all ships have been sunk, the program prints out a message that the game is over and tells how many shots were required.

Data Structures

We represent a ship by means of tuples

```
(row, column, horizontal, length, hits)
```

where:

- `row` and `column` are integers between 0 and 9 identifying, respectively, the row and column of the square of the top-left corner of the ship
- `horizontal` is a Boolean value equal to `True` if the ship is placed horizontally and `False` if placed vertically
- `length` is an integer between 1 and 4 representing the length of the ship
- `hits` is a set of tuples of the form `(row, column)` containing all the squares occupied by the ship that were hit

We represent a fleet by means of a list

```
[ship1, ship2, ....]
```

of ships. Note that during the game, a fleet will contain 10 ships (which may be intact, hit, or sunk), however, when the computer places the ships randomly, it is convenient to start with an empty list and iteratively expand it by adding ships.

Required Functions

- `is_sunk(ship)` -- returns Boolean value, which is `True` if `ship` is sunk and `False` otherwise
- `ship_type(ship)` -- returns one of the strings `"battleship"`, `"cruiser"`, `"destroyer"`, or `"submarine"`identifying the type of `ship`
- `is_open_sea(row, column, fleet)` -- checks if the square given by `row` and `column` neither contains nor is adjacent (horizontally, vertically, or diagonally) to some ship in `fleet`. Returns Boolean `True` if so and `False` otherwise
- `ok_to_place_ship_at(row, column, horizontal, length, fleet)`-- checks if addition of a ship, specified by `row, column, horizontal`, and `length` as in `ship` representation above, to the `fleet` results in a legal arrangement (see the figure above). If so, the function returns Boolean `True` and it returns `False` otherwise. This function makes use of the function `is_open_sea`
- `place_ship_at(row, column, horizontal, length, fleet)` -- returns a new fleet that is the result of adding a ship, specified by `row, column, horizontal`, and `length` as in `ship` representation above, to `fleet`. It may be assumed that the resulting arrangement of the new fleet is legal
- `randomly_place_all_ships()` -- returns a fleet that is a result of a random legal arrangement of the 10 ships in the ocean. This function makes use of the functions `ok_to_place_ship_at` and `place_ship_at`
- `check_if_hits(row, column, fleet)` -- returns Boolean value, which is `True` if the shot of the human player at the square represented by `row` and `column` hits any of the ships of `fleet`, and `False` otherwise
- `hit(row, column, fleet)` -- returns a tuple `(fleet1, ship)` where `ship` is the ship from the fleet `fleet that` receives a hit by the shot at the square represented by `row` and `column`, and `fleet1` is the fleet resulting from this hit. It may be assumed that shooting at the square `row, column` results in of some ship in `fleet`
- `are_unsunk_ships_left(fleet)` -- returns Boolean value, which is `True` if there are ships in the fleet that are still not sunk, and `False` otherwise
- `main()` -- returns nothing. It prompts the user to call out rows and columns of shots and outputs the responses of the computer (see *General Idea of Assignment*) iteratively until the game stops. Our expectations from this function: (a) there must be an option for the human player to quit the game at any time, (b) the program must never crash (i.e., no termination with Python error messages), whatever the human player does. Note that there is an indicative implementation of `main()` to help you start working, but it does not satisfy the expectations above and you should improve or entirely redo it.

Additional functions

You can add any additional functions if you need. We recommend also testing your new functions, if they are easily testable, but doing that would not affect your mark.

Extra Libraries/Packages

You can use any standard Python libraries if you need. They must be installable in both pip and conda (by running `pip install ...` and `conda install ...`). The installation must not require any manual configuration.

Extension

In addition, you can implement the visualisation of the game. Our expectations of the high-quality visualisation are as follows:

- The new state of the ocean and fleet are presented to the human player each time after he/she shoots
- The rows and columns of the ocean are numbered
- The squares that have never been shot at are clearly indicated (i)
- The squares that have been shot at but not resulted in a hit (nothing was there) are clearly indicated (ii)
- The squares containing a ship (of unknown type) that has been hit but not yet sunk are clearly indicated (iii)
- The ships that were sunk are clearly indicated, as well as their type (iv)

The following is one example of high-quality visualisation:

```
  0 1 2 3 4 5 6 7 8 9
 ---------------------
0| . . . . . . . . . .
1| . C C C . * - . . *
2| . . . . . . . . . .
3| . * . . * - . . . .
4| . - . . . . . S . .
5| . * . . . . . . . -
6| . . - . . - . . . *
7| . . . - . - . . . *
8| . - . . . . . . . .
9| . . . . . . . . . .
```

In this example, (i) from the list above are indicated by `.`, (ii) by `-`, (iii) by `*` and (iv) by a letter specifying the type of a ship (S - submarine, C - cruiser)

Files and Repo

- `readme.md` -- this file (together with several picture files)
- `battleships.py` contains the implementation of all the required functions (see *Required functions*)
- `test_battleships.py` contains unit tests for all *testable* functions from the set of required functions in the Pytest format. Testable are all the required functions except for `randomly_place_all_ships()` and `main()`

The files above with basic templates are provided in the repo. You must edit `battleships.py` and `test_battleships.py` and return in your submission. It is required to use Pytest for tests in this project. Moreover, if you provide the visualisation extension

(see *Extension*), add the file `extension.py` to your submission. The game with visualisation must be startable by executing `extension.py`.

You can add additional files if you need them, provided that the requirements above on `battleships.py`, `test_battleships.py`, and `extension.py` are satisfied.