README.md

Assignment 6 - Battleship

BACKGROUND

Battleship is a famous two-player board game. Each player is given 5 ships: a Carrier, a Battleship, a Cruiser, a Submarine, and Destroyer. The objective of Battleship is to be the first player to sink the other player's 5 ships. The first player to sink the opponent's ships wins. Each ship occupies a specific number of positions on the board: Carrier (5 spaces), Battleship (4 spaces), Cruiser (3 spaces), Submarine (3 spaces), and Destroyer (2 spaces). Before the game begins, each player places the 5 ships on the board. Once the game starts, the ships cannot be moved. Neither player can see the placement of the opponent's ships. The real-world Battleship game provides two boards for each player. A lower (horizontal) board is where the player places their 5 ships. An upper (vertical) board is where the player records guesses have been made to track where to guess in future turns.

The following rules restrict ship placement:

- All 5 ships must be placed on the board.
- Ships can only be placed vertically or horizontally on the board; diagonal placement is not allowed.
- No ship can hang off the board.
- No ship can overlap another on the board.

Both players are supplied with red and white pegs. Players take turns calling out board coordinates. When a coordinate called out by a player represents a position on the opponent's board where a ship has been placed, the opponent responds "hit". Otherwise, the opponent responds "miss". The player will mark a *miss* with a white peg when a *miss* occurs. A red peg is used when a *hit* occurs. The opponent places a red peg on the ship when a *hit* occurs. The opponent makes no peg placement after a *miss*.

When a ship has red pegs in all holes, the ship is sunk. The opponent must announce "hit and sunk" in this case. The game ends when all of the ships of one player have all been sunk.

CS1114 Battleship is a simplified version of this game. Only one player is involved in the game. Ship placement follows the rules above but is managed by the program (not by a player). The player does not need to manage placement of ships, therefore, the board only reflects the results of the coordinate guesses of an attack. The player can only miss on 20 (`MAX_MISSES`) coordinate guesses during the course of the game. If the player sinks all ships before exhausting this number of guesses, the player wins. Otherwise, the player loses.

This assignment will give you the opportunity to implement Battleship using an Objected-Oriented Programming (OOP) approach. `Ship` and `Game` are Python classes. The different types of ships are instances (objects) of the `Ship` class. A single instance of the `Game` class is used in the program. Your task will be to implement the `Ship` and `Game` classes for the Battleship program. You will then use objects of these classes to implement the program as described above.

Let's get started.

TABLE OF CONTENTS

1. Step 0 - Read the README
2. Instructions
   i. Step A - Enable Ship initialization
   ii. Step B - Enable Game initialization
   iii. Step C - Identify in-bounds and non-overlapping ship positions
   iv. Step D - Identify valid ship placement positions

   v. Step E - Add ships to the game
   vi. Step F - Get player's guess
   vii. Step G - Check for successful hit
   viii. Step H - Update the game based on user's guessed position
   ix. Step I - Check for a completed game
   x. Step J - Allow user to play again
   xi. Step K - Complete program
3. Evaluation
4. Assignment Submission

STEP 0 - READ THE README

Before writing a single line of Python code for this assignment, you are **strongly** encouraged to read this document in its entirety. The document includes key details about the expected implementation of your program. Diving into the implementation based solely on the description of the program above is likely to result in following a path towards your program's implementation that fails to meet the requirements of this assignment.

If you would like to consume this document in an alternative format, the README has also been converted to a PDF which is available in this repository.

INSTRUCTIONS

A number of program constants that will be useful in implementing this program are included at the top of the starter code. Be sure to use these constants where appropriate rather than magical numbers and character literals in your program. This program will focus on completing the implementation of two classes `Game` and `Ship`. The `play_battleship()` function for this program has been implemented for you as the goal of the assignment is to focus on Object-Oriented Programming (OOP) concepts. Make a function call to the `play_battleship()` function in the `main()` function when the class implementations are complete in order to run the program.

Step A - Enable Ship initialization

Method: `Ship.__init__(self, name, start_position, orientation)`

```python
def __init__(self, name, start_position, orientation):
    """Creates a new ship with the given name, placed at start_position in the
    provided orientation. The number of positions occupied by the ship is determined
    by looking up the name in the SHIP_SIZE dictionary.

    :param name: the name of the ship
    :param start_position: tuple representing the starting position of ship on the board
    :param orientation: the orientation of the ship ('v' - vertical, 'h' - horizontal)
    :return: None
    """
```

The first task for this assignment is to enable initialization of `Ship` instances.

A `Ship` has 4 data attributes: `name` (a `str`), `positions` (a `dict`) representing the board positions occupied by the ship and their *hit* status, and `sunk` (a `bool`). The `positions` `dict` uses a `tuple` key where the first item in the `tuple` is the `row` (`A` - `J`) and the second item is the `column` (`0` - `9`) of one position which the `ship` occupies on the board. The values in the `positions` dict is a `bool` indicating whether or not the ship has been *hit* at this position.

The `Ship` **init**() method has 3 parameters (other than self): `name`, `start_position`, and `orientation`. For the purposes of this assignment, the name of the ship will be one of `carrier`, `battleship`, `cruiser`, `submarine`, or `destroyer`. Using the value referenced by the `name` parameter, the number of positions occupied by the ship can be looked up in the `SHIP_SIZE` `dict` that has been provided for you. `start_position` is a tuple where the first item in the `tuple` is the `row` (`A` - `J`) and the second item is the `column` (`0` - `9`) of the first position of the ship. You can assume that the `start_position` is a valid position on the board. `orientation` is a string with one of two possible values: `h` or `v`. Using the number of positions occupied by the ship, the value referenced by `start_position`, and the value referenced by `orientation`, the `positions` dictionary can be defined.

When `orientation` is `v`, all positions occupied by the ship are in the **same column** of the board. For example, a `carrier` (occupying 5 positions) and a `start_position` of `('B',3)` will occupy positions `('C',3)`, `('D',3)`, `('E',3)`, and `('F',3)` as seen on the board below. The positions occupied by the `carrier` are represented by the `^` character. The use of `^` for showing the ship's location is for visualization purposes only and does not need to be implemented in the program.

```
  0 1 2 3 4 5 6 7 8 9
A . . . . . . . . . .
B . . . ^ . . . . . .
C . . . ^ . . . . . .
D . . . ^ . . . . . .
E . . . ^ . . . . . .
F . . . ^ . . . . . .
G . . . . . . . . . .
H . . . . . . . . . .
I . . . . . . . . . .
J . . . . . . . . . .
```

In this case, the `positions` attribute of the ship references a `dict` object with 5 keys (`('B', '3)`, `('C', 3)`, `('D', 3)`, `('E', 3)`, `('F', 3)`) each with a value of `False` after the `__init__()` method is invoked.

```
>>> ship = Ship("carrier", ('B', 3), 'v')
>>> print(ship.positions)
{('B', 3): False, ('C', 3): False, ('D', 3): False, ('E', 3): False, ('F', 3): False}
```

The `ord()` and `chr()` functions will prove useful in specifying the position entries for ship's in the `v` orientation.

When `orientation` is `h`, all positions occupied by the ship are in the **same row** of the board. For example, a `battleship` (occupying 4 positions) with a `start_position` of `('F',5)` will also occupy positions `('F', 6)`, `('F', 7)`, and `('F', 8)` as seen on the board. The positions occupied by the `battleship` are represented by the `^` character. Again, the use of `^` for showing the ship's location is for visualization purposes only and does not need to be implemented in the program.

```
  0 1 2 3 4 5 6 7 8 9
A . . . . . . . . . .
B . . . . . . . . . .
C . . . . . . . . . .
D . . . . . . . . . .
E . . . . . . . . . .
F . . . . . ^ ^ ^ ^ .
G . . . . . . . . . .
H . . . . . . . . . .
I . . . . . . . . . .
J . . . . . . . . . .
```

In this case, the `positions` attribute (a dictionary) for the ship instance contains 4 keys (`('F',5)`, `('F', 6)`, `('F', 7)`, `('F', 8)`) each with a value of `False` after invoking the `__init__()` method.

```
>>> ship = Ship("battleship", ('F', 5), 'h')
>>> print(ship.positions)
{('F', 5): False, ('F', 6): False, ('F', 7): False, ('F', 8): False}
```

The value passed to the `name` parameter is valid when it is one of the keys stored in the `SHIP_SIZE` dictionary. Be sure to assign the value referenced by the `name` parameter to the `name` attribute of the `Ship` object being initialized.

The ship's `sunk` attribute is initialized to `False` in the `__init__()` method definition.

Once the `Ship` class' `__init__()` method has been implemented, try submitting your `battleship.py` file to Gradescope to see if your implementation passes the **Step A** tests. Once your program passes the **Step A** tests, proceed to **Step B**.

Step B - Enable Game initialization

Method: `Game.__init__(self, max_misses)` and `Game.initialize_board(self)`

```
def __init__(self, max_misses = MAX_MISSES):
    """ Creates a new game with max_misses possible missed guesses.
    The board is initialized in this function and ships are randomly
    placed on the board.

    :param max_misses: maximum number of misses allowed before game ends
    """



def initialize_board(self):
    """Sets the board to it's initial state with each position occupied by
    a period ('.') string.

    :return: None
    """
```

A `Game` object keeps track of the state of a single game. This includes the existing ships, the board, and position guesses that have been made by the player. The `Game`'s `__init__()` method has two main responsibilities: initializing the board and placing the ships on the board. You will implement two separate methods for performing these tasks and call the methods within the definition of the `Game`'s `__init__()` method. The `Game` `__init__()` method has 1 parameters (excluding `self`). The value of the `max_misses` parameter is the value used to set the `max_misses` attribute of the `Game` object being instantiated. Use the program constant `MAX_MISSES` as the default parameter value for `max_misses`. Initialize the `max_misses` attribute for the `Game` object to the value passed to the `max_misses` parameter in the `__init__()` method definition. Initialize the `Game` object's `ships` attribute to an empty `list` in the `__init__()` method definition. Finally, initialize the `Game` object's `guesses` attribute to an empty `list` in the `__init__()` method definition.

Step B focuses on initializing the board. The `intialize_board()` has no parameters (besides `self`). After this method is executed, the `board` attribute of a `Game` object can displayed to the player. The `board` attribute is a Python `dict`. The keys are uppercase letters (`A` - `J`). The corresponding value for each key is a `list` of 10 strings. The initial value for each item in the `list` is the period (`'.'`) string. Therefore, the `board` dictionary is a dictionary containing 10 `str` \ `list` key-value pairs.

For clarification, the `board` should be initialized in such a way that accessing the `C` row of the board will return a 10 item `list` where each item in the `list` is the period string (`.`).

```
>>> game.board['C']
['.', '.', '.', '.', '.', '.', '.', '.', '.', '.']
```

The initial board (a `dict`) can be visualized with the row values (the `dict` keys) referencing a list of periods (the `dict` values).

A → | . | . | . | .| . | . | . | . | . | . |
B → | . | . | . | .| . | . | . | . | . | . |
C → | . | . | . | .| . | . | . | . | . | . |
...
I → | . | . | . | .| . | . | . | . | . | . |
J → | . | . | . | .| . | . | . | . | . | . |

Implement the `Game` `__init__()` method such that it initializes the `board` attribute to an empty `dict`. Implement the `Game` `initialize_board()` method to populate the `board` `dict` as described above. After implementing the `Game` `initialize_board()` method, call the `initialize_board()` method on the `Game` instance within the `Game` `__init__()` method definition. Implementation of the `Game` `__init__()` method continues in **Step C** and **Step D**.

After defining the methods `__init__()` and `initialize_board()` in the `Game` class, try submitting your `battleship.py` file to Gradescope to see if your implementation passes the **Step B** tests. Once your program passes the **Step B** tests, proceed to **Step C**.

Step C - Identify in-bounds and non-overlapping ship positions

Methods: `Game.in_bounds(self, start_position, ship_size, orientation)` and `Game.overlaps_ship(self, start_position, ship_size, orientation)`

```
def in_bounds(self, start_position, ship_size, orientation):
    """Checks that a ship requiring ship_size positions can be placed at start position.

    :param start_position: tuple representing the starting position of ship on the board
    :param ship_size: number of positions needed to place ship
    :param orientation: the orientation of the ship ('v' - vertical, 'h' - horizontal)
    :return status: True if ship placement inside board boundary, False otherwise
    """


def overlaps_ship(self, start_position, ship_size, orientation):
    """Checks for overlap between previously placed ships and a potential new ship
    placement requiring ship_size positions beginning at start_position in the
    given orientation.

    :param start_position: tuple representing the starting position of ship on the board
    :param ship_size: number of positions needed to place ship
    :param orientation: the orientation of the ship ('v' - vertical, 'h' - horizontal)
    :return status: True if ship placement overlaps previously placed ship, False otherwise
    """
```

This step focuses on implementation of methods for determining valid ship placements. Recall these rules of the game.

1. Ships cannot be outside of the boundaries of the board.
2. A newly placed ship cannot overlap with a ship that has already been placed.

Therefore, in order to place ships, we want to define two methods which are helpful for identifying valid ship placements.

The `in_bounds()` method has 4 parameters (including `self`). `start_position` is the initial position on the `board` that the ship will occupy on successful placement. `ship_size` is the number of positions on the `board` required to place the ship. `orientation` is the horizontal (`h`) or vertical (`v`) orientation being considered for the ship. This method returns `True` when a ship occupying `ship_size` positions starting at `start_position` in the given `orientation` would not exceed the boundaries of the `board`. For example, consider a randomly generated position, `('D', 9)`, when trying to place a `destroyer` (requiring 2 positions) on the board below.

```
  0 1 2 3 4 5 6 7 8 9
```

```
A . . . . . . . . . .
B . . . . . . . . . .
C . . . . . . . . . .
D . . . . . . . . . s
E . . . . . . . . . .
F . . . . . . . . . .
G . . . . . . . . . .
H . . . . . . . . . .
I . . . . . . . . . .
J . . . . . . . . . .
```

The `s` on the board represents the `start_position`. Notice, that a horizontal (`h`) placement of the `destroyer` is not possible because the ship would fall outside of the bounds of the board. Therefore, a call to `in_bounds()` with these parameter values returns `False`.

```
>>> pos = ('D',9)
>>> size = 2
>>> orient = 'h'
>>> game.in_bounds(pos, size, orient)
False
```

However, a call to `in_bounds()` when `orientation` is passed the value `v`, the return value of `in_bounds()` is `True`.

```
>>> pos = ('D',9)
>>> size = 2
>>> orient = 'v'
>>> game.in_bounds(pos, size, orient)
True
```

When a ship placement in the `v` orientation is considered, `start_position` represents the smallest row value in lexicographic ordering of the rows that are occupied by the ship. When a ship placement in the `h` orientation is considered, `start_position` represents the smallest column value of the columns that are occupied by the ship.

As `ship` s are placed on the `board`, some placements will not be possible because such placements would cause `ship` positions to overlap. The `overlaps_ship()` method has the same 4 parameters that are defined for the `in_bounds()` method. Consider the board below:

```
  0 1 2 3 4 5 6 7 8 9
A . . . . . . . . . .
B . . . . . . . . . .
C . . . . . . . . . .
D . ^ ^ ^ ^ ^ . . . .
E . . . . . . . . . .
F . . . . . . . . . .
G . . . . . . . . . .
H . . . . . . . . . .
I . . . . . . . . . .
J . . . . . . . . . .
```

The `^` s represent a ship occupying positions `('D',1)`, `('D',2)`, `('D',3)`, `('D',4)`, and `('D',5)`. An attempt to place a `battleship` at position `('B',2)` in vertical (`v`) orientation is not allowed because it would cause two ships to occupy the same position. Therefore, the corresponding method invocation of `overlaps_ship()` would return `True`.

```
>>> pos = ('B',2)
>>> size = 4
```

```
>>> orient = 'v'
>>> game.overlaps_ship(pos, size, orient)
True
```

However, for the board above, the `overlaps_ship()` method invocation for the same `ship_size` and `start_position` using the horizontal (`h`) orientation would return `False` .

```
>>> pos = ('B',2)
>>> size = 4
>>> orient = 'h'
>>> game.overlaps_ship(pos, size, orient)
False
```

The `overlaps_ship()` method requires checking the keys of the `positions` dictionary of previously placed ships that have been added to the `ships` list of the `Game` object for possible overlapping board positions. Keep in mind that if a ship has been placed in the game, the ship will already have been added to the `ships` list for the `Game` object. You may be asking, when did we place ships on the `board` . The answer is that we have not yet implemented this part of the program. Placing ships on the board is functionality that will be implemented in `Step D` and `Step E` . However, for implementing the `overlaps_ship()` method, assume that the `Game` instances `ships` list contains ships 0 or more ships that have been placed in a particular game. You have defined all of the data attributes required to implement the `in_bounds()` and `overlaps_ship()` methods despite not yet having implemented the functionality for placing ships and adding the ships to the `ships` list of the `Game` object.

After defining the methods `in_bounds()` and `overlaps_ship()` methods in the `Game` class, try submitting your `battleship.py` file to Gradescope to see if your implementation passes the **Step C** tests. Once your program passes the **Step C** tests, proceed to **Step D**.

Step D - Identify valid ship placement positions

Method: `Game.place_ship(self, start_position, ship_size)`

```python
def place_ship(self, start_position, ship_size):
    """Determines if placement is possible for ship requiring ship_size positions placed at
    start_position. Returns the orientation where placement is possible or None if no placement
    in either orientation is possible.

    :param start_position: tuple representing the starting position of ship on the board
    :param ship_size: number of positions needed to place ship
    :return orientation: 'h' if horizontal placement possible, 'v' if vertical placement possible,
        None if no placement possible
    """
```

Let's now use `in_bounds()` and `overlaps_ship()` in order to determine if a ship of a given `ship_size` can be placed on the `board` at a given `start_position` . Again, the rules for placing ships:

1. Ships cannot be outside of the boundaries of the board.
2. A newly placed ship cannot overlap with a ship that has already been placed.

Both conditions need to be satisfied in order to allow a ship to be placed. The `place_ship()` method has 2 parameters (excluding `self`). `start_position` is the first position on the `board` where the `ship` will be placed for a potential placement. `ship_size` is the number of positions on the `board` required to place the ship. This method first checks that a ship placement at `start_position` for a ship requiring `ship_size` positions in the horizontal (`h`) orientation is in the bounds of the board and does not overlap any previously placed ship. In the case that a horizontal placement is possible, the method invocation returns `h` . In the case that the ship cannot be placed in the horizontal orientation, the method checks that a ship placement at `start_position` for a ship requiring `ship_size` positions in the vertical (`v`) orientation is in the bounds of the board and does not overlap any previously placed ship. In the case that a horizontal placement is possible, the method invocation returns `v` . In the case that the ship cannot be placed in the vertical orientation, the method returns `None` .

Consider the board below:

```
  0 1 2 3 4 5 6 7 8 9
A . . . . . . . . . .
B . . . . . . . . . .
C . . . . ^ ^ ^ ^ . .
D . . . . . . . . . .
E . . . . . . . . . .
F . . . . . . . . . .
G . . . . . . . . . .
H . . . . . . . . . .
I . . . . . . . . . .
J . . . . . . . . . .
```

A ship is currently occupying positions `('C', 4)` , `('C', 5)` , `('C', 6)` , and `('C', 7)` . Attempting to place a ship requiring 3 positions with a `start_position` of `('A', 6)` in the vertical (`v`) orientation does not exceed the boundaries of the board. However, this placement is not possible because the ship would overlap a previously placed ship at position `('C',' 6)` .

Using a horizontal (`h`) orientation, the placement is possible. This is the case because the placement is completely inside of the boundaries of the board **and** does not overlap the ship that was previously placed.

```
>>> pos = ('A',6)
>>> size = 3
>>> game.place_ship(pos, size)
'h'
```

Therefore, the `place_ship()` method invocation returns `h` , indicating that this ship can be placed horizontally.

Consider a different configuration of the game:

```
  0 1 2 3 4 5 6 7 8 9
A . . . . . . . . . .
B . . . . ^ . . . . .
C . . . . ^ . . . . .
D . . . . ^ . . . . .
E . . . . ^ . . . . .
F . . . . . . . . . .
G . . . . . . . . . .
H . . . . . . . . . .
I . . . . . . . . . .
J . . . . . . . . . .
```

A ship is currently occupying positions `('B', 4)` , `('C', 4)` , `('D', 4)` , and `('E', 4)` . Attempting to place a ship requiring 3 positions with a `start_position` of `('C', 3)` in the horizontal (`h`) orientation does not exceed the boundaries of the board. However, this placement is not possible because the ship would overlap a previously placed ship at position `('C', 4)` .

Using a vertical (`v`) orientation, the placement is possible. This is the case because the placement is completely inside of the boundaries of the board **and** does not overlap the ship that was previously placed.

```
>>> pos = ('C',2)
>>> size = 3
>>> game.place_ship(pos, size)
'v'
```

Therefore, the `place_ship()` method invocation returns `v` , indicating that this ship can be placed vertically.

Finally, consider the following case:

```
  0 1 2 3 4 5 6 7 8 9
A . . . . . . . . . .
B . . . . ^ . . . . .
C . . . . ^ . . . . .
D . . . . ^ . . . . .
E . . . . ^ . . . . .
F . . . . . . . . . .
G . . ^ ^ ^ ^ ^ . . .
H . . . . . . . . . .
I . . . . . . . . . .
J . . . . . . . . . .
```

A ship is currently occupying positions `('B', 4)` , `('C', 4)` , `('D', 4)` , and `('E', 4)` . Another ship is occupying `('G', 2)` , `('G', 3)` , `('G', 4)` , `('G', 5)` and `('G', 6)` . Attempting to place a ship requiring 3 positions with a `start_position` of `('E', 2)` in the horizontal (`h`) orientation does not exceed the boundaries of the board. The same is true of attempting to place a ship requiring 3 positions with a `start_position` of `('E', 2)` in the vertical (`v`) orientation. However, neither placement is possible. A vertically oriented placement would overlap another ship at `('G', 2)` . A horizontally oriented placement would overlap another ship at `('E', 4)` .

```
>>> pos = ('E',2)
>>> size = 3
>>> ret_val = game.place_ship(pos, size)
>>> print(ret_val)
None
```

Therefore, a method invocation of `place_ship()` in this situation would return neither `h` or `v` but would instead return `None` .

Your implementation of `place_ship()` will utilize method invocations of `in_bounds()` and `overlaps_ship()` as these methods implement the logic required for determining whether or not a ship of a given size, a proposed starting position, and a given orientation can be properly placed.

Once the `place_ship()` method has been implemented for the `Game` class, try submitting your `battleship.py` file to Gradescope to see if your implementation passes the **Step D** tests. Once your program passes the **Step D** tests, proceed to **Step E**.

Step E - Add ships to the game

Method: `Game.create_and_place_ships(self)` Method to update: `Game.__init__(self, max_misses)`

```python
def create_and_place_ships(self):
    """Instantiates ship objects with valid board placements.

    :return: None
    """
```

Having implemented methods to identify where a ship can be located, the focus of this step is creating the game's ships and placing the ships on the board. The method `create_and_place_ships()` implements this functionality. This method has no other parameters besides `self`. All of the information required to create and place ships is contained within the `Game` object itself.

The game of Battleship would not be very interesting if all of the ships are clustered together and placed in easily predictable locations. Randomly placing ships creates a game where identifying the location of ships is more difficult. The `get_random_position()` helper function has been implemented for you. This function randomly generates a position within the bounds of the board as a potential starting location for a ship that needs to be placed on the board. This method will return a `tuple` such as `('E', 2)` representing a position on the board. In this example, `E` is a `string` representing the row label and `2` is an `int` representing the column label for a position. The function declaration and docstring are as follows:

```python
def get_random_position():
    """Generates a random location on a board of NUM_ROWS x NUM_COLS

    :return: tuple representing a random position on the board
    """
```

The placement of ships should be performed in decreasing order of the number of positions occupied by a ship. The `_ship_types` attribute of the `Game` class is a list that contains the names of the ships in decreasing order of size. The process for placing a ship proceeds as follows:

1. For each ship needing a placement, a random position is generated using `get_random_position()`.
2. A placement for the ship at that location is attempted using the game's `place_ship()` method. The size of the ship can be obtained by looking up the ship name in the `SHIP_SIZES` `dict`.
3. If a valid placement is possible, the ship is placed. Otherwise, a new random position is generated and a new placement attempted.

This process continues until a valid position and orientation is found for the ship. Once a valid position and orientation is identified, a new `Ship` object is instantiated using the `start_position` and `orientation`. Finally, the newly created ship is added to the `Game` object's `ships` list attribute.

When an invocation of the `create_and_place_ships()` method is complete, all `Ship` objects for the game will have been instantiated and added to the game. Add a method invocation for `create_and_place_ships()` to the Game class' `__init__()` method to ensure that ships are created and placed every time a new game is created.

After implementing the `create_and_place_ships()` method in the `Game` class, try submitting your `battleship.py` file to Gradescope to see if your implementation passes the **Step E** tests. Once your program passes the **Step E** tests, proceed to **Step F**.

Step F - Get player's guess

Method: `Game.get_guess(self)`

```python
def get_guess(self):
    """Prompts the user for a row and column to attack. The
    return value is a board position in (row, column) format

    :return position: a board position as a (row, column) tuple
    """
```

At this point in the implementation of your program, `Game` and `Ship` objects can be properly instantiated. However, the full Battleship implementation requires additional functionality to execute a Battleship game. The program must prompt the user for a guess. This functionality is implemented in the `Game` class' `get_guess()` method. The `get_guess()` method requires no additional parameters besides `self`. The method prompts the user for a `row` and `column` representing a location on the board that the user wants to attack. The method ensures that a valid `row` and `column` are provided. A valid `row` is one that is in the character range `A-J`. A valid `column` is one that is in the integer range `0-9`. When **both** a valid `row` and `column` are provided, a (`row`, `column`) tuple representing this position is returned by the method where `row` is a single-character string and `column` is an integer.

```
>>> game.get_guess()
Enter a row: o
Enter a row: j
Enter a row: B
Enter a column: -3
Enter a column: 10
Enter a column: 4
('B', 4)
```

Notice that the prompt `Enter a row:` is repeated until a valid `row` value is provided. The prompt `Enter a column:` is not shown until a valid `row` has been provided. Once a valid `column` is provided, the method returns the position tuple.

After implementing the `get_guess()` method in the `Game` class, try submitting your `battleship.py` file to Gradescope to see if your implementation passes the **Step F** tests. Once your program passes the **Step F** tests, proceed to **Step G**.

Step G - Check for successful hit

Method: `Game.check_guess(self, position)`

```
def check_guess(self, position):
    """Checks whether or not position is occupied by a ship. A hit is
    registered when position occupied by a ship and position not hit
    previously. A miss occurs otherwise.

    :param position: a (row,column) tuple guessed by user
    :return: guess_status: True when guess results in hit, False when guess results in miss
    """
```

Once a valid position has been provided by the user, the program needs to check if a ship occupies the position guessed by the user. When a ship occupies the guessed position **and** the position has not been previously hit, this guess should be registered as a *hit*. Registering a hit requires updating the value for the `ship`'s `positions` `dict` to `True` at the position (the key of the `dict`) where the hit occurs. When the position is empty **or** the position has been previously hit, the guess should be registered as a *miss*. The `check_guess()` method has one parameter (besides `self`): the (row,column) `position` tuple guessed by the user. The `check_guess()` method returns `True` when the guess is a *hit* and `False` when the guess is a *miss*.

Consider the case that a destroyer is located at positions `('A', 7)` and `('A', 8)`. Checking a guess of position `('A',7)` returns `True`.

```
>>> guess = ('A', 7)
>>> game.check_guess(guess)
True
```

If the same position `('A', 7)` is checked again, the method returns `False`.

```
>>> guess = ('A', 7)
>>> game.check_guess(guess)
```

```
False
```

A position (`('D', 8)` , for example) not occupied by a ship also returns `False` .

```
>>> guess = ('D', 8)
>>> game.check_guess(guess)
False
```

Additionally, when a hit is registered by a guess that results in all of the positions of a ship having been successfully attacked, the ship is considered to be sunk. In this case, the `ship` 's `sunk` attribute is set to `True` . Finally, `You sunk the <ship_name>!` is output to the screen where `<ship_name>` is the name of the `ship` that was sunk.

Continuing with the previous example, a guess of `('A',8)` passed to `check_guess()` results in a hit. This hit results in all positions on the destroyer successfully being attacked resulting in the destroyer being sunk, output indicating that the destroyer was sunk, and the `check_guess()` method invocation returning `True` .

```
>>> guess = ('A', 8)
>>> game.check_guess(guess)
You sunk the destroyer!
True
```

After implementing the `check_guess()` method in the `Game` class, try submitting your `battleship.py` file to Gradescope to see if your implementation passes the **Step G** tests. Once your program passes the **Step G** tests, proceed to **Step H**.

Step H - Update the game based on user's guessed position

Method: `Game.update_game(self, guess_status, position)`

```
def update_game(self, guess_status, position):
    """Updates the game by modifying the board with a hit or miss
    symbol based on guess_status of position.

    :param guess_status: True when position is a hit, False otherwise
    :param position:  a (row,column) tuple guessed by user
    :return: None
    """
```

After checking the position guessed by the user, the game needs to be updated. When the user's guess results in a *hit*, the `board` is updated. When the user's guess is a *miss*, the `board` attribute and the `guesses` attribute are both updated. The `update_game()` method has 3 parameters (besides `self`). `guess_status` is the value returned by `check_guess()` indicating whether a guess resulted in a *hit* or a *miss*. `guess_status` is `True` when `position` results in a *hit*. `guess_status` is `False` when `position` results in a *miss*. `position` is the guess provided by the user.

`position` is a tuple with a `row` and `column` value. `row` references the key in the `Game` object's `board` `dict` . `column` references the index of the `list` that is the value in the `board` `dict` for the key `row` . `update_game()` sets the value at the index `column` of the list assigned to the key `row` in `board` to `x` when `guess_status` is `True` . `update_game()` sets the value at the index `column` of the list assigned to the key `row` in `board` to `o` when `guess_status` is `False` . An `x` provides a visual reminder that a *hit* occurred at `position` when the board is displayed for each round of the game. An `o` provides a visual reminder that a *miss* occurred at `position` to discourage the user from guessing the same position again.

`position` is added to the `guesses` list attribute of the `Game` object when `guess_status` is `False` . The board was represented using this diagram in a previous step:

A → | . | . | . | .| . | . | . | . | . | . |
B → | . | . | . | .| . | . | . | . | . | . |
C → | . | . | . | .| . | . | . | . | . | . |
...
I → | . | . | . | .| . | . | . | . | . | . |
J → | . | . | . | .| . | . | . | . | . | . |

Consider when `update_game()` is invoked with `guess_status` passed the value `True` and `position` passed the value `('C',0)`. This method invocation would change the board to reflect a successful hit on a ship:

A → | . | . | . | .| . | . | . | . | . | . |
B → | . | . | . | .| . | . | . | . | . | . |
C → | x | . | . | .| . | . | . | . | . | . |
...
I → | . | . | . | .| . | . | . | . | . | . |
J → | . | . | . | .| . | . | . | . | . | . |

In a subsequent round, a guess of `('J',2)` when `guess_status` is `False` would result in the following board representation:

A → | . | . | . | .| . | . | . | . | . | . |
B → | . | . | . | .| . | . | . | . | . | . |
C → | x | . | . | .| . | . | . | . | . | . |
...
I → | . | . | . | .| . | . | . | . | . | . |
J → | . | . | o | .| . | . | . | . | . | . |

A method invocation of `update_game()` that is passed `False` for the `guess_status` parameter when the position on the board has been updated in a previous round does not update the board again.

After implementing the `update_game()` method in the `Game` class, try submitting your `battleship.py` file to Gradescope to see if your implementation passes the **Step H** tests. Once your program passes the **Step H** tests, proceed to **Step I**.

Step I - Check for a completed game

Method: `Game.is_complete(self)`

```python
def is_complete(self):
    """Checks to see if a Battleship game has ended. Returns True when the game is complete
    with a message indicating whether the game ended due to successfully sinking all ships
    or reaching the maximum number of guesses. Returns False when the game is not
    complete.

    :return: True on game completion, False otherwise
    """
```

At this point, a large portion of the Battleship program's functionality is in place. A game can be initiated, a guess can be provided by the user, a guess can be checked for a successful *hit* or *miss*, and the board can be updated. These events will happen during each round of a Battleship game. However, this program does not yet contain any functionality for determining when a game is complete. A game is complete when one of two events has occurred:

1. All ships on the board are sunk
2. The number of missed guesses by the user reaches the maximum allowed number of misses

The `is_complete()` method has no parameters (besides `self`). This method checks the current state of the game and returns `True` when one of the two conditions above is met. If neither condition applies, the `is_complete()` returns `False`. In the case that the game is complete because the user successfully sunk all ships, `is_complete()` outputs `YOU WIN!`. When the game is complete due to the user reaching the maximum number of misses, `is_complete()` outputs `SORRY! NO GUESSES LEFT.`.

Consider the case where the board is in the following state:

```
  0 1 2 3 4 5 6 7 8 9
A x x . . . . . . . .
B . . . . . . . . . .
C . . . x x x x . . .
D . . . . . . . . . .
E x x x . . . . . . .
F . . . . . . . . . .
G x x x x x . . . . .
H x x x . . . . . . .
I . . . . . . . . . .
J . . . . . . . . . .
```

This board indicates all ships at every position have sustained a hit. A method invocation of `is_complete()` for this game

```
>>> game.is_complete()
YOU WIN!
True
```

shows the output `YOU WIN!` with a return value of `True`. However, in a situation where the user has exceeded the max number of guesses

```
>>> game.is_complete()
SORRY! NO GUESSES LEFT.
True
```

the output is `SORRY! NO GUESSES LEFT.` with a return value of `True`. In most rounds, these conditions are not met, so an invocation of `is_complete()` at the beginning of a round results simply in a return value of `False`:

```
>>> game = Game()
>>> game.is_complete()
False
```

When the `is_complete()` method in the `Game` class has been defined, try submitting your `battleship.py` file to Gradescope to see if your implementation passes the **Step I** tests. Once your program passes the **Step I** tests, proceed to **Step J**.

Step J - Allow user to play again

Function: `end_program()`

```python
def end_program():
    """Prompts the user with "Play again (Y/N)?" The question is repeated
    until the user enters a valid response (Y/y/N/n). The function returns
    False if the user enters 'Y' or 'y' and returns True if the user enters
    'N' or 'n'.

    :return response: boolean indicating whether to end the program
    """
```

Multiple Battleship games can be played in one session of the program. The user determines how many games will be played. Therefore, at the completion of a game, the program requires a way to determine whether to start a new game or end the program. The `end_program()` function has no parameters. The function returns `True` when the user indicates a desire to end the program. `end_program()` returns `False` when the user indicates a desire to play again. This function uses the prompt `Play again (Y/N)?` to determine if the user wants to continue playing or end the program. The user must enter `Y` or `y` to play again. The user enters `N` or `n` to end the program. Any other input from the user causes the prompt `Play again (Y/N)?` to be displayed again until a valid input values is provided by the user.

```
>>> end_program()
Play again (Y/N)? x
Play again (Y/N)? 8
Play again (Y/N)? y
False
```

The example above demonstrates that while the user is prompted to enter `Y` to play again, `y` is also accepted. When the user enters `n` or `N` the function will return `True`.

```
>>> end_program()
Play again (Y/N)? n
True
```

The `end_program()` is a stand-alone function that is not defined inside of a class.

When the `end_program()` function has been defined, try submitting your `battleship.py` file to Gradescope to see if your implementation passes the **Step J** tests. Once your program passes the **Step J** tests, proceed to **Step K**.

Step K - Complete program

Due to the focus of this assignment being the use of Object-Oriented Programming, the `main()` and `play_battleship()` functions have been provided in the starter code file. For this step, you simply need to call the `play_battleship()` function from the `main()` function and remove the `pass` statement. If **Steps A - J** have been implemented correctly, the program is complete and a fully functioning Battleship program is the result!

A few guesses demonstrating the execution of a game up to the point of sinking a carrier is exhibited below:

```
Let's Play Battleship!

  0 1 2 3 4 5 6 7 8 9
A . . . . . . . . . .
B . . . . . . . . . .
C . . . . . . . . . .
D . . . . . . . . . .
E . . . . . . . . . .
F . . . . . . . . . .
G . . . . . . . . . .
H . . . . . . . . . .
I . . . . . . . . . .
J . . . . . . . . . .

Enter a row: A
Enter a column: 3

  0 1 2 3 4 5 6 7 8 9
A . . . o . . . . . .
B . . . . . . . . . .
```

```
C . . . . . . . . . .
D . . . . . . . . . .
E . . . . . . . . . .
F . . . . . . . . . .
G . . . . . . . . . .
H . . . . . . . . . .
I . . . . . . . . . .
J . . . . . . . . . .

Enter a row: B
Enter a column: 6

  0 1 2 3 4 5 6 7 8 9
A . . . o . . . . . .
B . . . . . . o . . .
C . . . . . . . . . .
D . . . . . . . . . .
E . . . . . . . . . .
F . . . . . . . . . .
G . . . . . . . . . .
H . . . . . . . . . .
I . . . . . . . . . .
J . . . . . . . . . .

Enter a row: C
Enter a column: 0

  0 1 2 3 4 5 6 7 8 9
A . . . o . . . . . .
B . . . . . . o . . .
C o . . . . . . . . .
D . . . . . . . . . .
E . . . . . . . . . .
F . . . . . . . . . .
G . . . . . . . . . .
H . . . . . . . . . .
I . . . . . . . . . .
J . . . . . . . . . .

Enter a row: A
Enter a column: 0

  0 1 2 3 4 5 6 7 8 9
A o . . o . . . . . .
B . . . . . . o . . .
C o . . . . . . . . .
D . . . . . . . . . .
E . . . . . . . . . .
F . . . . . . . . . .
G . . . . . . . . . .
H . . . . . . . . . .
I . . . . . . . . . .
J . . . . . . . . . .

Enter a row: D
Enter a column: 5

  0 1 2 3 4 5 6 7 8 9
A o . . o . . . . . .
B . . . . . . o . . .
C o . . . . . . . . .
D . . . . . o . . . .
```

```
E . . . . . . . . . .
F . . . . . . . . . .
G . . . . . . . . . .
H . . . . . . . . . .
I . . . . . . . . . .
J . . . . . . . . . .

Enter a row: F
Enter a column: 4

  0 1 2 3 4 5 6 7 8 9
A o . . o . . . . . .
B . . . . . . o . . .
C o . . . . . . . . .
D . . . . . o . . . .
E . . . . . . . . . .
F . . . . x . . . . .
G . . . . . . . . . .
H . . . . . . . . . .
I . . . . . . . . . .
J . . . . . . . . . .

Enter a row: F
Enter a column: 3

  0 1 2 3 4 5 6 7 8 9
A o . . o . . . . . .
B . . . . . . o . . .
C o . . . . . . . . .
D . . . . . o . . . .
E . . . . . . . . . .
F . . . x x . . . . .
G . . . . . . . . . .
H . . . . . . . . . .
I . . . . . . . . . .
J . . . . . . . . . .

Enter a row: F
Enter a column: 5

  0 1 2 3 4 5 6 7 8 9
A o . . o . . . . . .
B . . . . . . o . . .
C o . . . . . . . . .
D . . . . . o . . . .
E . . . . . . . . . .
F . . . x x x . . . .
G . . . . . . . . . .
H . . . . . . . . . .
I . . . . . . . . . .
J . . . . . . . . . .

Enter a row: F
Enter a column: 2

  0 1 2 3 4 5 6 7 8 9
A o . . o . . . . . .
B . . . . . . o . . .
C o . . . . . . . . .
D . . . . . o . . . .
E . . . . . . . . . .
F . . x x x x . . . .
```

```
G . . . . . . . . . .
H . . . . . . . . . .
I . . . . . . . . . .
J . . . . . . . . . .

Enter a row: F
Enter a column: 6
You sunk the destroyer!

  0 1 2 3 4 5 6 7 8 9
A o . . o . . . . . .
B . . . . . . o . . .
C o . . . . . . . . .
D . . . . . o . . . .
E . . . . . . . . . .
F . . x x x x x . . .
G . . . . . . . . . .
H . . . . . . . . . .
I . . . . . . . . . .
J . . . . . . . . . .

Enter a row:
```

EVALUATION

This assignment will use Gradescope's Autograder tool. Therefore, you will receive immediate feedback when submitting this assignment. You should strive to have your submitted code pass all of the tests run by the Gradescope Autograder by the time you have completed **Step K**.

It is just as important to write clean and readable code as it is to write correct and functional code. A portion of your grade on this assignment will be based on how well you follow certain coding conventions. Make sure to follow the standards discussed in class, and before you submit your assignment, take a minute to review your code to check for stylistic issues like those mentioned below.

Comments

To make your program easier to read, you should add comments before (docstrings) and inside your functions/methods to make your intention clearer. Good comments give the reader a clue about what a function/method does and, in some cases, how it works.

Be sure to include header comments (including your name) for the submitted file.

Function/method design

Function/method names should include verbs indicating what is accomplished by the function/method. Most of the functions/methods in this assignment have been named for you. However, if you add any additional functions/methods, be sure to name these functions/methods appropriately:

- function/method names should include verbs
- all words in a function/method name should be written in lower-case letters
- multiple words in a function/method name should be separated by underscores

Functions/methods should be decomposed such that the length of each is small. This is an important part of proper decomposition. There is a bit of an art to this but you should ensure that your functions/methods solve well-defined sub-problems. The function/method definitions should not, in general, have 10s of lines of codes. This is especially true for the programs written in this class.

Function/method use

- Functions/methods that are defined for the program should be used. If program behavior captured by a function/method is duplicated by code in another part of the program, your submission will be penalized.
- Functions/methods should not include function/method calls to themselves (recursive definitions). This likely means that you need to use a loop in your program and are not doing so. Such recursive function/method definitions will be penalized.

Variable names

Variable naming conventions are as follows:

- names should be descriptive
- all characters in the name should be written in lowercase (with the exception of constant variables)
- multiple words should be separated by underscores ('_').

Spacing

Indentation should be consistent in the files that are submitted. The convention is that each block of indented code is indented using four spaces. Your program should follow this convention.

Spaces should be used between operators and operands in the expressions written in your code.

Magic Numbers

Your program should not contain any constant numeric values. Constant variables are to be defined and used in place of hard-coding numeric constants into the program.

ASSIGNMENT SUBMISSION

You should submit the following file on Gradescope (do not include any files not included in the list below):

- **battleship.py**

The Autograder will assign points to passed tests. These results should be used as a guide for keeping you on track towards an implementation of this assignment that meets all of the requirements laid out above.

Good luck!