Battleship Lite

Krisztina Tesenyi

Code overview

```python
# Player class is created with the Player's name as a property.
class Player:
    def __init__(self) -> None:
        self.name = ""

    # greet player, name input check
    def greet_player(self):
        while True:
            try:
                self.name = input("Please enter your name :")
                if (bool(re.fullmatch('^[a-zA-Z0-9 _-]{2,25}+$', self.name))):
                    break
                else:
                    raise ValueError
            except ValueError:
                print('The name is not valid!')
                continue
        print(f'Hello {self.name}, let\'s start the game!')


# Game control
def main():
    # Create player object and call its methods to greet the player
    player = Player()
    player.greet_player()
```

Result

```
(venv) kriszta@DESKTOP-FF6D3JS:/mnt/c/Users/krisz
Please enter your name :*Kriszta
The name is not valid!
Please enter your name :Kriszta T
Hello Kriszta T, let's start the game!
```

The Player class is defined. The initial property is the name of the player, which is an empty string.

- greet_player method is for asking for the name of the player and printing a greeting text including the player's name. To make sure that the given user name is valid a regex pattern was defined ('^[a-zA-Z0-9 _-]{2,25}+$'). The name can contain lower and upper cases (a-zA-Z), digits (0-9), space, underscore, and hyphens (_-). The length of the user name needs to be between 2 and 25 characters ({2,25}).
  - re.fullmatch function was used, so only fully matching user names can be accepted. Fullmath returns a match object, that was converted into a Boolean value, so error handling can be done using the expression. (bool(re.fullmatch('^[a-zA-Z0-9 _-]{2,25}+$', self.name)))
  - try-except block embedded in a while loop was used to handle ValueErrors in the name. The while loop was necessary, so a name can be asked until a valid input name is not given. Try block include asking for input from the player and an if statement examining if the name input is fully matches with the regex pattern if so, the code breaks out the while loop and prints the greeting text. In any other case, a ValueError is raised which is handled in the except block. Except block print out a message to the player informing that invalid name was provided and continue with the while loop next iteration.
- main function includes the Player object creation and greet_player method was called to see the result of the code.

Code overview

```python
from tabulate import tabulate

# First a Game map class is created, properties rows, cols also a game_map
# list is initialised which will contain the map. The map will be a state map
# it is filled with zeros as a start, zero means no ship.


class Game_map:
    def __init__(self, rows=10, cols=10) -> None:
        self.rows = rows
        self.cols = cols
        self.game_map = []

    def create_game_map(self):
        self.game_map = [[0]*self.cols for _ in range(self.rows)]

    def print_game_map(self):
        headers = 'ABCDEFGHIJ'
        print(tabulate(self.game_map, headers=headers, tablefmt='fancy_grid',
              showindex=range(1, self.rows + 1)))


def main():
    game_map = Game_map()
    game_map.create_game_map()
    game_map.print_game_map()


if __name__ == "__main__":
    main()
```

Result of the code

	A	B	C	D	E	F	G	H	I	J
1	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0

The Game_map class is created. Properties: rows (10), cols (10), also a game_map list is initialised. The list will contain the map, which will be a state map.

- create_game_map method builds up the map, using a combination of value repetition and a for loop. It puts zeros 10 times (which is how many columns are in the map ([0]*self.cols), in all 10 rows (for _ in range(self.rows)) in the 2D list. The map is filled with zeros as a start, zero means nothing (no ship, no shot, no hit) is on a certain coordinate.
- print_game_map method prints out the map using the tabulate package. A header and index are added to the map, and a tablemft format is fancy_gird.
- main function will include code that is relevant for controlling the game. A Game_map object was created called game_map and methods were called to build and print the game_map for test.
- if __name__ == "__main__":
  main()

  This code block allows execution of the file when it runs as a script, but not when it is imported as a module.

```python
class Ships:
    def __init__(self, ship_type, length, map_id) -> None:
        self.ship_type = ship_type
        self.length = length
        self.life = length
        self.map_id = map_id
        self.is_alive = True

    def get_ship_type(self):
        return self.ship_type

    def get_length(self):
        return self.length

    def get_life(self):
        return self.life

    def decrease_life(self):
        self.life -= 1

    def get_map_id(self):
        return self.map_id


class Carrier(Ships):
    def __init__(self, map_id=5) -> None:
        super().__init__('carrier', 5, map_id)


class Battleship(Ships):
    def __init__(self, map_id=4) -> None:
        super().__init__('battleship', 4, map_id)


class Cruiser(Ships):
    def __init__(self, map_id=3) -> None:
        super().__init__('cruiser', 3, map_id)


class Submarine(Ships):
    def __init__(self, map_id=1) -> None:
        super().__init__('submarine', 3, map_id)


class Destroyer(Ships):
    def __init__(self, map_id=2) -> None:
        super().__init__('destroyer', 2, map_id)
```

Code overview

To place ships on the game map ships need to be created. The best way to achieve that is to define a parent Ships class first, that has the following properties that all different ships have and will inherit:

- self.ship_type = ship_type (i.e.: Carrier)
- self.length = length (i.e.: Carrier is a 5-grid long ship)
- self.life = length (i.e.: All ship has their life (which is equal to their length), and loses 1 life after every hit)
- self.map_id = map_id (i.e.: All ships have a unique map id. It comes to play when we want to know which ship is hit so we can decrease its life or which ship is where on the map.)
- self.is_alive = True (It is also important to know if a given ship is alive or not, to win the game all ships need to be destroyed. Initially, is_alive value is True, and once all life is lost, the life value is 0, then is_alive gets the value False. After all ship is_alive value is False the player wins the game.)

- get methods were created for ship_type, length, life and map_id. These will be asked during the game many times from the ship objects.
- decrease_life method was created to decrease the life value of the ship and will be called once the ship gets hit.

The next step is to create classes for the individual ships that inherit from the Ships' parent object.

class Carrier(Ships): <- Carrier class inherit from Ships class
def __init__(self, map_id=5) -> None: <- Carrier init only define map_id as it can be changed during object creation. It can be useful if we want to add 2 carriers for example.
super().__init__('carrier', 5, map_id) <- inherited properties from Ships class, initial values defined ('carrier', 5, map_id).

Future development options

- Ask user input for the size of the map (rows, cols), they are defined as parameters of the Game_map class, so it can be easily added as an option in the future.
- Add more ships to the game than five especially if the game map is bigger than 10*10. As classes were built, we can create as many ship objects as we want to put on our map. And because there is a map_id for each ship, we can define unique ids for all of them even if they are the same types when we create an instance.