Thought Process

Grid Creation and Obstacle Placing

I set my first objective to be simple, and that was to implement the grid creation and the obstacle placing functionalities. These 2 functionalities will be implemented in one single class as they are closely interrelated with each other

The task of creating the grid was simple, I created a class called Grid which would handle the grid creation and the obstacle placing. In this class I created a private instance variable 'matrix' which would have X rows and Y columns. This is created upon initializing an object of the class with the variables x and y.

Implementing the obstacle placing: My approach was to simply accept a list of coordinates from the user. However, this step was prone to errors on the user's side, therefore I implemented a helper function within my class that would throw an error if the user entered an invalid coordinate.

List of possible errors:

- The input is not a list or tuple
- The list or tuple is empty
- The coordinates within the list were invalid if:
    - They weren't numbers
    - They weren't a list or a tuple
    - They had a length longer than 2 (Each coordinate can have only one x and one y)
    - The coordinate was outside the range of the matrix.

With respect to the last point about the coordinates being in range, I decided to make the whole input invalid if any one of the inputs were wrong. This maintains simplicity, however it is not that big of a task pick all the valid coordinates only.

I have successfully managed everything I mentioned above while sticking to the SOLID principles. Creating the grid and placing the obstacles are closely interrelated therefore put those 2 functionalities in one single class.

CHANGELOG:

Instead of printing error messages I created a class that would help raise exceptions instead. This will help us have more control over what to do when an exception occurs.

Rover

Now I shall create a class that initializes the starting position if the rover.

I implemented a class called Validity and took the coordinate validation method straight out of the Grid_Obstacle class. Used this Validity class in both the grid_Obstacle(I removed the original method) and Rover to make the code more neat.

However, after successfully implementing valid start position, grid creation and obstacle placing by using methods from the validator class. I realized that I could make it a parent "Base" class and use inheritance to inherit its methods directly into grid_obstacle class and rover_start class.

Successfully implemented the above, this covers an extremely good demonstration of encapsulation and inheritance.

I implemented some helper functions outside my class in the Rover code to make my main function neater and debug errors better. I have also added some extra functions to my base class that helps with ease of life such as get orientation with word which returns the direction as a full word. I added a 1 second sleep in a few places where it felt like the output was too quick and to introduce a "Mars rover operation" feel to things.

I decided to save the Grid and pass it onto the file where the rover navigation takes place. I also sent a list of obstacles and the position of obstacles. I decided not to implement a live rover but instead as the problem states will accept a list of directions from the user and move the rover and display its final location and mention its orientation. I have to decide whether to just stop the rover when an obstacle is detected or skip the command.

Note: Decided to skip the command and move on to the next when an obstacle is detected

Moving Mechanism:

I decided to create a MarsRover class that has the following instance attributes: start positions(x and y), Gridsize(a new grid class that has x and y attribute to show its size), obstacles and direction. 5 in total.

Move Command:

If the rover is facing north I would decrement the x axis. Before doing this I would check if there is an obstacle. If there is an obstacle I will (skip or terminate) the moving of the rover. If it is facing east the move command would increment the y-axis. This pattern would follow for all 4 directions.

Turn left and Turn right:

Simple logic, based on which direction the rover is facing, we update the direction. If facing east and it turns right is passed, it will turn south.

Short Summary

- I basically used the classes: GridManager, GridObstacle and RoverStart as classes that do **VISUALIZATION AND EXCEPTION HANDLING** of the user inputs.
- The MarsRover class accepts the values we obtained after thorough exception handling from the above classes and handles all the movement logic of the rover.

OOP Principles:

- I have successfully demonstrated encapsulation at all places through my use of private attributes and instance attributes
- I have used Inheritance in the following places:
    - RoverStart is a child class of GridObstacle which in turn is a child class of GridManager
    - The whole command.py code is a perfect example of how interface leads to polymorphism
- Polymorphism is evident in the way I am calling 3 different possible execute commands with the same line. Refer commands.py code for clarity

SOLID Principles: I have stuck to most of the SOLID principles as well. All classes are responsible for one thing. I used abstract classes. Made the interface simple and non-complicated. The classes are open for extension as well (OCP).

END OF THOUGHT PROCESS