CSC148 Exercise 2

(Due: May 17, 10:00 pm on Markus)

Remember, no late submissions accepted. Make sure to submit timely. Do not leave it for the last minute - internet connection may go down, server may be overloaded - there are many things that can go wrong - unfortunately they do not count as valid reasons for accommodation.

In this exercise, you'll work on applying what you've learned about inheritance, a fundamental type of relationship between classes, to extend your work from last week. Specifically, you will see how to use inheritance to design classes for different types of vehicles, all of which support the same basic operations, but do so in their own unique ways.

Learning Goals:

- Practice inheritance.
- Implement a class based on an abstractly defined set of methods.

Cars and other vehicles

Recall that last week, you helped design and implement the `Super Duper` ridesharing system, which tracks and moves cars across a 2-D grid.

To build on your work from last week, you will now allow `Super Duper` to support not just cars, but other vehicles as well.

Various vehicles do share common properties (such as `position` and `fuel`); also they do share methods that do *the same thing* in different manners. For example, a car *moves* to a destination `(x, y)` along the horizontal and vertical lines of the grid, whereas a helicopter can also move *diagonally*. Although mutating the position of any vehicle (`Car, Helicopter, ...`) to a new positions amounts to simply set the coordinates, the computation of the fuel needed to accomplish the move, differs in different vehicles. This implies certain methons need be defined *abstractly* in the `Vehicle` class and *implemented differently* in different subclasses of `Vehicle`.

To do this, we have defined a `Vehicle` *interface* - please read its documentation carefully. We have also modified the `SuperDuperManager` class (minus dispatch) to make use of this interface (our code likely looks very similar to your solution from last week).

You have two responsibilities here:

- Update the `Car` class you implemented last week to now implement the Vehicle interface. All characteristics of Cars from last week are applied to this week.
- Create two new classes `Helicopter` and `UnreliableMagicCarpet`, both of which implement `Vehicle`, according to the following descriptions:

  `Helicopter`: starts at position (3, 5), the Super Duper launchpad, with a specified amount of fuel. It uses 1 unit of fuel per unit of distance, but it can travel diagonally, and always goes in a straight line between its current position and target destination. Like `Car`, it does not move and uses no fuel if it does not have enough fuel to complete the journey. After each move, round the amount of fuel down to the nearest integer.

  `UnreliableMagicCarpet`: starts at a random position (x, y) where both x and y are random integers between 0 and 10, inclusive. Unlike the other two vehicles, it does not use any fuel when it moves (so its fuel attribute never changes from its starting value). However, when told to move to position (x, y), it instead moves to a random position `(x + dx, y + dy)`, where `dx` and `dy` are random integers between -2 and 2, inclusive. (So some of the time it might move to the correct position, but it often doesn't.)

Warning: Please do not modify the `SuperDuperManager` class.

Keep the following points in mind when implementing your three classes:

A constructor for the superclass Vehicle is provided - you should use it inside all of the subclass constructors. We're expecting a certain public interface for the constructor of each class. See the SuperDuperManager's `add_vehicle` method for details.

The `fuel_needed` method is documented but left unimplemented in `Vehicle`: all of its subclasses are required to implement it. A default implementation of move is given in Vehicle. For each subclass, you should look carefully at whether this implementation makes sense, or whether to override.