CSC148 exercise#2, winter 2017
due: Sunday January 22nd, before 10 p.m.

In this exercise, you'll gain some practice designing and implementing a class, according to an English problem description. You will also practice using this class from another class, or, in other words, relating two classes through composition.

It's a good idea to read through this handout, the starter code, and the sample tests early, so that you understand what it is you will accomplish.

Save these files to your ex1 folder (inside exercises).

- ex1.py
- ex1_test.py

task 1: car simulation

In this task, you'll implement a basic car simulation, to track the positions and amounts of fuel of a set of cars in a city.

Super Duper is a new car ride-sharing service which uses a computer system to manage its autonomous vehicles. The managing system contains a collection of cars, each uniquely identified by a string. The manager can add a new car to the collection, move a car, and get the current location of a car.

When a Super Duper car is created in the system, it is given an initial amount of fuel (specified individually per car), and starts at position (0,0) on the grid, representing the Super Duper headquarters. Cars can move to a new point (any integer coordinates are allowed) on the grid, but only move in horizontal and vertical lines. Cars cannot travel diagonally.

When cars move, they lose 1 unit of fuel per 1 unit of distance moved. If you try to move a car farther than it can go with its remaining fuel, the car stays in its original position, and does not use any fuel. (In other words, this method will "fail silently." In general this is a bad practice, and we'll explore better ways of handling situations like this later in the course.) Cars can move multiple times, as long as they have enough fuel.

In the starter code, we have created a SuperDuperManager class, which keeps track of all the cars in the Super Duper system. It is responsible for creating new cars, looking up a particular car and retrieving its position or fuel information, and directing cars to move to new locations. Read through the docstrings of the SuperDuperManager class and its methods.

The class has an instance attribute _cars which you must use to complete the Car simulation. This is a "private" attribute; treat it like any other attribute for now. SuperDuperManager's methods correspond to the actions we just described, and you may not change their signatures.

your job(s)

1. design and implement a Car class, which represents a single car according to the above description.
2. Implement the methods of the SuperDuperManager class to make use of your Car class.

submission instructions

All work is done individually for this exercise. Submit the required file(s) electronically through the MarkUs submission system. The due date and time is a firm one. You can submit multiple times on MarkUs! Use this feature to submit early and often.

Before the deadline, make sure you've done all of the following: saved all changes, making sure your files are up to date run the sample tests (and made sure they all passed) added additional tests to ensure the correctness of your program, run python_ta.check_all() in your code (see "main" block at the bottom), and fixed all style errors that came up consulted the exercise marking scheme (see below) submitted your work on MarkUs

For more information about the software we're using in this course, check out these pages:

- python_ta quick start, and python_ta error messages
- hypothesis testing quick start

marking scheme

All exercises in this course will be automatically graded, and the mark for each one is broken down into the following two components:

Correctness (90%): this is based purely on number of tests passed. Please note that we'll run additional tests beyond the sample ones provided, so it is your responsibility to create your own tests to verify the correctness of your program. The number of provided tests will always be at least one-half of the total number of tests.

Style (10%): this is based on the number of errors reported by running python_ta.check_all() on your code. Every error results in a 1% deduction (up to 10 deductions total). Multiple errors of the same type are counted individually.

Please note that the starter code runs python_ta.check_errors(), which looks only for the most important potential errors in your code, to help you during debugging. You should change this to call python_ta.check_all() before your submission, as this is what we'll be doing during grading.

An important habit to get into is to use the tools that are provided to you. It is possible to be guaranteed a 60% on each exercise simply by running the sample tests and python_ta.check_all(), and fixing any problems before submission. We hope you take advantage of this!