CSC148 - Inheritance: Super Duper Manager Example

Lec3

Introduction

In this exercise, you're going to work on code for the *Super Duper* ride-sharing service, one with much more interesting vehicle options than Uber or Lyft! We model each vehicle's position on a 2-D grid using (x, y) integer coordinates. Here is a comparison of the three types of vehicle:

Type	Initial position	Moves to	Can move diagonally?	Fuel usage
car	(0, 0)	the desired location	✗	1 unit fuel per unit distance
helicopter	(3, 5), the launchpad	the desired location	✓	1 unit fuel per unit distance
unreliable magic carpet	a random location*	a random location*	✓	uses no fuel

*The initial location of an unreliable magic carpet is limited: x and y must each be between -10 and 10. And when an unreliable magic carpet moves, its new location is a random spot within 2 units horizontally and 2 units vertically of the desired location. For example, if the desired location is (10, 20), then the magic carpet location's x coordinate would be between 8 and 12, and its y coordinate would be between 18 and 22.

What to do

We have given you printed starter code on the attached sheet. Your tasks involve reading this code and answering questions about it, and then working on implementing the missing parts using inheritance.

1. What are the two classes defined in the starter code, and what additional classes will you need to create for this exercise? Identify any inheritance relationships between them.

   vehicle class and SuperDuperManager class.

   We need car, helicopter & magic carpet classes. those are child class of vehicle class.

2. Read the `Vehicle` class and answer these questions:

   (a) What instance attributes does every vehicle possess?

   position & fuel

   (b) Does every vehicle start with the same amount of fuel and the same position?

   No, it depends on how we initialize those.

   (c) Why is method `fuel_needed` not implemented?

   Because different tools have diff fuel cost, ~~equate different~~ we overwrite in child class.

   (d) After reading class `Vehicle`, what method(s) do you know for sure must be defined in each of its subclasses?

   fuel_needed.

   move.

3. Class `SuperDuperManager` keeps track of all the vehicles. Read the header and docstring for each of its methods. Write doctest examples for the class that use each method at least once. (If the docstrings are well written, you should *not* have to read method bodies to be confident that you know what each method does!)

4. Find the code that keeps track of all the vehicles.

   (a) What instance attribute is used to keep track of the vehicles? What is the type of this attribute?

   id — , ~~string~~ _vehicles : dict[str, vehicle]

   (b) Where is it initialized?

   ~~_vehicles : dict[str, Vehicle] or self._vehicles = {}~~ in the initizer.

   (c) Where is it updated?

   self.vehicles[id_] = ...

   in add_vehicle.

5. You are about to implement class `Car` (on a separate sheet of paper). Let's figure out the attributes.

   (a) If you just accept the initializer that `Car` inherits from class `Vehicle`, what instance attributes will every car possess?

   position & fuel

   (b) Does class `Car` need any other instance attributes?

   fuel_cost = 1

6. Now let's figure out the methods of class `Car`.

   (a) What method(s) does `Car` inherit?

   ~~fuel~~ fuel_needed & move

   ass __init__

   (b) Which of these inherited methods must it implement because its parent did not?

   fuel_needed

   (c) Which of these inherited methods must it override because the inherited implementation is not adequate? In these cases, should it call the parent class method as a helper, to get part of the work done?

   yes __init__ & fuelneeded

   def __init__(self, initial_fuel):

   vehicle.__init__(self, init, (x, y))

   ~~self._fuel_cost = fuel_cost~~

7. With all of this sorted out, the code is much easier to write. Implement class `Car`.

8. Use the same strategy as in questions 5 and 6 to implement class `Helicopter` and class `UnreliableMagicCarpet`.

```python
from __future__ import annotations
from math import sqrt
import random          # used to generate random numbers

class Vehicle:
    """An abstract class for a vehicle in the Super Duper system.

    Attributes:
    - position: The coordinates of this vehicle on a grid.
    - fuel: The amount of fuel remaining for this vehicle.

    Representation invariants:
    - fuel >= 0
    """
    position: tuple[int, int]
    fuel: int

    def __init__(self, initial_fuel: int,
                 initial_position: tuple[int, int]) -> None:
        """Initialize a new Vehicle with the given fuel and position.

        Precondition: initial_fuel >= 0
        """
        self.fuel = initial_fuel
        self.position = initial_position

    def fuel_needed(self, new_x: int, new_y: int) -> int:
        """Return how much fuel would be needed to move to the given position.

        Note: the amount returned may be larger than self.fuel,
        indicating that this vehicle may not move to the given position.
        """
        raise NotImplementedError

    def move(self, new_x: int, new_y: int) -> None:
        """Move this vehicle to a new position.

        Do nothing if this vehicle does not have enough fuel to move to the specified position.
        """
        needed = self.fuel_needed(new_x, new_y)
        if needed <= self.fuel:
            self.position = (new_x, new_y)
            self.fuel -= needed
```

class .

```python
class SuperDuperManager:
    """A class responsible for keeping track of all vehicles in the system.
    Attributes:
    - _vehicles: Maps a string that uniquely identifies a vehicle to the corresponding Vehicle object.
                 For example, _vehicles['car1'] would be a Vehicle object with the id_ 'car1'.
    """
    _vehicles: dict[str, Vehicle]

    def __init__(self) -> None:
        """Initialize a new SuperDuperManager.

        There are no vehicles in the system when first created.
        """
        self._vehicles = {}

    def add_vehicle(self, vehicle_type: str, id_: str, fuel: int) -> None:
        """Add a new vehicle with the given type, id_, and fuel to the system.

        Do nothing if there is already a vehicle with the given id.

        Preconditions:
        - vehicle_type in {'Car', 'Helicopter', 'UnreliableMagicCarpet'}
        - fuel >= 0
        """
        # Check to make sure the identifier isn't already used.
        if id_ not in self._vehicles:
            if vehicle_type == 'Car':
                self._vehicles[id_] = Car(fuel)
            elif vehicle_type == 'Helicopter':
                self._vehicles[id_] = Helicopter(fuel)
            elif vehicle_type == 'UnreliableMagicCarpet':
                self._vehicles[id_] = UnreliableMagicCarpet(fuel)

    def move_vehicle(self, id_: str, new_x: int, new_y: int) -> None:
        """Move the vehicle with the given id.

        The vehicle called <id_> should be moved to position (<new_x>, <new_y>).
        Do nothing if there is no vehicle with the given id,
        or if the corresponding vehicle does not have enough fuel to move.
        """
        if id_ in self._vehicles:
            self._vehicles[id_].move(new_x, new_y)

    def get_vehicle_position(self, id_: str) -> tuple[int, int] | None:
        """Return the position of the vehicle with the given id.

        Return a tuple of the (x, y) position of the vehicle.
        Return None if there is no vehicle with the given id.
        """
        if id_ in self._vehicles:
            return self._vehicles[id_].position
        return None

    def get_vehicle_fuel(self, id_: str) -> int | None:
        """Return the amount of fuel of the vehicle with the given id.

        Return None if there is no vehicle with the given id.
        """
        if id_ in self._vehicles:
            return self._vehicles[id_].fuel
        return None
```