CSC148, Assignment #1
ride-sharing simulation
due February 19th, 10 p.m.

deadline to declare your assignment team: February 15th, 10 p.m.

overview

Car testers use crash-test dummies because the danger of human carnage is too great to use living subjects. Climatologists build computer models of changing meteorological patterns because they can't afford the decades or centuries to wait observing the actual patterns.

Computer simulations are great for exploring "what if" questions about scenarios before, or instead of, letting those scenarios play out in the real world. In this assignment, you'll create a simulation of a ride-sharing service to see the effect of different scenarios. We hope this is a safer and faster approach than having taxis, Uber, riders, and municipal politicians live through the scenarios on the streets.

This assignment will give you experience reading, understanding, designing and implementing several interacting classes to model prospective riders with varying amounts of patience, available drivers starting from various locations, and dispatchers trying to match up the riders and drivers. You'll also work on implementations of classes to run the simulation and monitor the results, and provide utilities such as sequences to hold objects and a class to represent locations on a grid.

simulation overview

The simulation consists of four key entities: Riders, drivers, the dispatcher, and the monitor. Riders request rides from their current location to a destination. Drivers drive to pickup and drop off riders. The dispatcher receives and satisfies requests from drivers for a rider, and from riders for a driver. The monitor keeps track of activities in the simulation, and when asked, generates a report of what happened during the simulation.

The simulation plays out on a simplified city grid, where the location of riders and drivers is the intersection they are closest to. An intersection is represented by a pair of positive integers: The number of an east/west street, and the number of a north/south street. For example 1,2 is the intersection of Street 1 East/West with Street 2 North/South.[1]

When a rider requests a driver, the dispatcher tries to assign a driver to the rider. If there is no driver available, the dispatcher keeps track of the rider request, waiting for a driver to become available. A rider will cancel their request if they have to wait too long for a pick up.

When a driver requests a rider, the dispatcher assigns a waiting rider, if any. If this is the driver's first request, the dispatcher registers the driver in its fleet. Once registered, a driver never unregisters.[2]

And so it goes. There is clearly a connection between how long riders are waiting to be picked up and how many drivers are available and waiting to be assigned riders. Our simulation can measure how this

[1] In an advanced simulation you might use an on-line mapping API with information about traffic congestion, street directions, and so on. This simulation is simplified in many ways, but you are welcome to extend it in second-year, or even over the summer.

[2] Of course, in the real world or a more complex simulation, drivers would come and go as freely as riders do.

connection affects rider waiting times and driver earnings. It will need to monitor all appropriate events: riders requesting rides, being picked up (or possibly cancelling first), then being dropped off; and drivers requesting to be assigned riders

rider class details

A rider has a unique identifier, an origin, a destination and a status. The rider's status is one of waiting to be picked up, cancelled, or satisfied (after they have been picked up). A rider also has a patience attribute: The number of minutes the rider will wait to be picked up before they cancel their ride.

driver class details

A driver has a unique identifier, a location, and, possibly, a destination. A driver knows their car's speed; for simplicity this speed is constant. A driver can determine how long it will take to travel from their current location to a destination by calculating the Manhattan distance and dividing it by the car's speed. This calculation is rounded to the nearest integer.

If a driver does not have a destination, the driver is idle. When a driver first becomes idle, the driver requests a rider. If assigned a rider, the driver drives to the rider's location. If the rider does not cancel before the pickup, the driver then picks up the rider and drives to the destination. When the ride starts, the driver learns the identity of the rider and the rider's destination.

dispatcher class details

The dispatcher keeps track of both available drivers and waiting riders, based on riders requesting rides and drivers announcing their availability. However, the dispatchers don't make this information public.

When a rider requests a ride, the dispatcher finds the driver who reports that they can pick up the rider the fastest, and assigns that rider to the driver. If there is no driver available, the dispatcher places the rider on a waiting list, waiting for a driver to become available.

If a rider gets fed up waiting, they tell the dispatcher that they are cancelling their ride request, and the dispatcher removes them from the waiting list. However, the rider may have already been assigned to a driver, who will arrive at the rider's location to find that the rider has gone.

When a driver announces her/his availability to drive, the dispatcher registers the driver as part of the driving fleet (if the driver has just started to drive for the shift). If there are one or more riders waiting, the dispatcher assigns the longest-waiting rider to this available driver.

event class details

Simulation events model what happens, and when. Each event occurs at some particular time, expressed as positive integers in some suitable units. This allows us to order events from earlier to later, using rich comparison operations `__lt__`, `__gt__`, etc. We model events with an abstract class `Event`, but we have to leave what an event does to subclasses that model particular types of events.

Notice that occurrence of an event may change several elements, and may schedule new events, in the simulation.

RiderRequest event: A rider requesting a driver will cause the dispatcher to try to assign the quickest-arriving available driver to that rider, or else put the rider on a waiting list. If a driver is assigned a rider,

they begin driving to the rider's location, and a pickup event is scheduled for the time they arrive. A cancellation event is also scheduled for the time the rider runs out of patience and cancels the request. The pickup succeeds or fails depending on whether it is scheduled earlier than the cancellation. In either case, the driver's destination is set to the rider's location.

DriverRequest event: A driver requesting a rider will cause the dispatcher to assign that driver the longest-waiting rider, if there are any riders. At the same time, a pickup event is scheduled for the time that the driver arrives at the rider's location. The rider will have already scheduled a cancellation, so again pickup success or failure comes down to whether the pickup or cancellation is scheduled earlier. Again, the driver's destination is set to the rider's location, whether the driver arrives for a successful pickup or finds that the rider has cancelled.

Cancellation event: A cancellation event simply changes a waiting rider to a cancelled rider, and doesn't schedule any future events. Of course, if the rider has already been picked up, then they are satisfied and can't be cancelled.

Pickup event: A pickup event sets the driver's location to the rider's location. If the rider is waiting, the driver begins giving them a ride and the driver's destination becomes the rider's destination. At the same time, a dropoff event is scheduled for the time they will arrive at the rider's destination, and the rider becomes satisfied. If the rider has cancelled, a new event for the driver requesting a rider is scheduled to take place immediately, and the driver has no destination for the moment.

Dropoff event: A dropoff event sets the driver's location to the rider's destination, and leaves the rider satisfied. The driver needs more work, so a new event for the driver requesting a rider is scheduled to take place immediately, and the driver has no destination for the moment.

All events are monitored, so that statistics can be reported at the end of the simulation. This means that, along with the event actions themselves, the event notifies the monitor of what is happening.

simulation class details

You are implementing an **event-driven simulation**. This means that your simulation is driven by a sequence of events, ordered according to the event sorting order. The simulation removes the highest priority event, has it carry out its actions, notifies the monitor of the event, and then returns to the sequence for the next highest priority event.

Of course, an interesting simulation will need an initial non-empty sequence of events to get things started. Since some events cause new events to be scheduled, the initial sequence may grow as a result of processing events.

Since we don't have any events that schedule new riders, and all riders will end up either cancelled or satisfied, we can assume that the sequence of events will eventually be empty. This ends the simulation.

When the simulation is finished, it returns a report of any statistics the monitor has gathered.

monitor class details

Our monitor records each event when it happens, then uses those records to calculate and report statistics.

All of our events fall into one of two categories: those initiated by a driver and those initiated by a rider. The monitor is notified of an activity — an event's time, category, description (e.g. dropoff, pickup, cancellation), identifier of the person initiating the event, and the location relevant to the event — and records this accordingly.

Making a report requires the monitor to consult its records and calculate the average rider waiting time, the average distance travelled by drivers (including both rides and getting to rider locations) and the average ride distance that each driver is carrying a rider.

location class details

Our simulation plays out on a simplified grid of city blocks. Each location is specified by a pair of non-negative integers, **(m, n)**, where **m** represents the number of blocks the location is from the left edge of the grid, and **n** is the number of blocks the location is from the bottom of the grid.

Since it is not, in general, possible to drive diagonally through blocks, the distance that determines how quickly a driver can travel between two locations is manhatten distance. This distance is the number of horizontal blocks that separate the two locations (never negative) plus the number of vertical blocks that separate the two locations (also never negative).

Sometimes our program will need to read a text file that includes locations, and turn those into location objects. This is a module-level function that doesn't need to be a method of a location class, but can simply be grouped with the class in the same module.

container class details

Several parts of our simulation will need sequences of objects that allow us to add new objects, remove already-stored objects, and determine whether the sequence is empty. For example, the simulation keeps track of a sequence of events that is ordered according to event comparison so that an event with the lowest timestamp is retrieved ahead of those with a higher timestamp. Dispatchers keep track of available drivers and waiting riders, and will need a similar (but not identical) tool.

We provide you with an abstract container class so that you can implement concrete subclasses when you need them. One concrete subclass you'll certainly need is a priority queue that maintains its elements in priority order (using rich comparison operators such as <, <=, etc.) so that the highest-priority element is always ready to be removed.

how to use starter code

This ride-sharing simulation needs several cooperating Python classes to do its job. Some classes are clients and, in turn, provide services, so the relationships can become involved. We want you to gain maximum experience from doing several activities that computer scientists do:

1. reading and comprehending existing code in order to understand its relationship to code you must implement;
2. reading and comprehending API-only (i.e. interface) code in order to implement the body of the code;
3. reading and comprehending client code, in order to implement the code for classes that provide(s) the service(s) it requires;
4. reading and comprehending skeleton code with comments (those that use # characters) guiding you to write your own code

The starter code falls into one or more of these learning activities:

container.py Instructors provide abstract class `Container`.
Notice that the methods raise `NotImplementedError`, and must be implemented in each subclass. Instructors also provide subclass `PriorityQueue`, except they provide only the API for the `add()` method.

event.py: Instructors provide abstract class `Event` which has several subclasses:

RiderRequest: Provided by the instructors;

DriverRequest: Skeleton code, with comments;

Cancellation, Pickup, Dropoff: Instructors provide the class name and client code that uses these classes.

The module-level function `create_event_list` has skeleton code, with comments to guide students.

monitor.py Instructors provide all except methods `_average_total_distance` and `_average_ride_distance`, for which we provide the API.

rider.py Instructors provide a class name and (elsewhere) client code that uses it.

dispatcher.py Instructors provide the API.

driver.py Instructors provide the API.

location.py Instructors provide the API.

simulation.py Instructors provide skeleton code with comments.

events.txt Instructors provide an example initial event list. Note: Successfully running simulation.py with this file is no guarantee of a correct implementation.

We suggest that you begin with `location.py`, `container.py`, `rider.py`, `driver.py`, and `dispatcher.py`, testing your work as you go so that you are confident in it as you proceed. Then tackle `event.py`, `simulation.py`, and `monitor.py`.

declaring your assignment team

You may do this assignment alone or with one other student. Your partner(s) may be from any section of the course at UTM. You must declare your team (even if you are working solo) using the MarkUs online system.

Navigate to the MarkUs page for the assignment and find "Group Information". If you are working solo, click "work individually". If you are working with another student:

First: one of you needs to "invite" the other to be partners, providing MarkUs with their utorid.

Second: the invited student must accept the invitation.

Important: there must be **only** one inviter, and the other group member accepts **after** being invited.

To accept an invitation, find "Group Information" on the Assignment page, find the invitation listed there, and click on "Join".

submitting your work

Submit the following files on MarkUs by 10 p.m. on February 19:

- `container.py`
- `dispatcher.py`
- `driver.py`
- `event.py`
- `location.py`
- `monitor.py`
- `rider.py`
- `simulation.py`

Click on the "Submissions" tab near the top. Click "Add a New File" and either type a file name or use the "Browse" button to choose one. Then click "Submit". You can submit a new version of a file later (before the deadline, of course); look in the "Replace" column.

Only one team member should submit the assignment. Because you declared your team, both of you will get credit for the work.