CSE 231, Fall 2007

Programming Project 10

Assignment Overview

In this assignment you will create and use classes, as well as create an event-driven simulation. The project description is long mostly because we tell you a lot of details about how to actually do it.

This assignment is worth 60 points (6.0% of the course grade) and must be **completed and turned in before 11:59 on Monday, November 19, 2007.**

Question

The International Center has four ATM machines. When students are waiting to use a machine, should they form four lines or one line?

Background

When considering a question like this one, one must decide a measure for comparison. A reasonable measure in this case is: average waiting time in line. The question can then be rephrased as:

> To minimize average waiting time in line for four ATM machines, should students form one line or four?

To make the problem easier, we will not consider the cases of two or three lines—one could argue that they are neither interesting nor practical cases. **To make this assignment easier we will require a simulation of one line with simulation of four lines earning five points of extra credit.**

Next, how does one develop and support an answer? A common technique is to simulate the situation, and we will use what is know as "event-driven" simulation—that is, we will identify events and let them drive the simulation (details provided below).

Another important issue is how do you simulate students' arrival at the ATM machines? Fortunately, research shows that arrivals that fit a Poisson distribution provide reasonable answers to this class of question. You do not need to know about Poisson distributions because we will provide a file of data which fits that distribution. Generating Poisson data in Python is trivial, but providing a common data file will make it easier for you to develop a correct program, make grading easier, and give you one less thing to worry about.

Event-driven simulation

Simulations are based on time, and event-driven simulation is based on events that are considered in order of the time when the events happen. If event 'A' will happen at time t = 4 and event 'D' will happen at time t = 2, we will process event 'D' first (at time t = 2) before we process event 'A'. We can maintain a list of events sorted by the time when they will happen. When we need the next event to process, we will find it at the beginning of the sorted list. We can pop it off the list, and then process it (whatever "process" means—explained below).

Whenever we add a new event to the list, we will resort the list so that the list is always kept sorted. We can do that by appending the new event to the end of the list, and then sort it.

What events can happen at our ATM machines? There are two:

- Arrival: an arrival event (which we will label as 'A') is the event when a person arrives at the line so that person gets into line at the end (append to the list of people in line).
- Departure: a departure event (which we will label as 'D') is the event when a person is finished at the ATM and departs.

One will reasonably ask: isn't there an event when someone gets to the front of the line and leaves the line to go to an available ATM? Yes, but that event happens when an ATM becomes available which also happens to be exactly when someone left the ATM—that is, it happens when a 'D' event happens.

Another important thing to consider is the time any individual spends at the ATM. One person might spend a couple of minutes making a quick withdrawal, while another person might spend fifteen minutes fumbling around and doing multiple transactions. That time is called the person's "service time" and it is the time that that person will spend at the ATM.

To start the simulation we will read in arrival data from a file ('arrivals.txt'). Each line of the file will contain two integers: an arrival time (the time when this person arrives at the end of the line) and a service time (the length of time that this person will spend at the ATM when they eventually get their turn at an ATM). To keep things simple, time will be an integer that starts at zero—we'll call it the "wallClockTime" because it represents the time that would be on a clock on a wall. A line in the file might be "15 5" which means that this person arrives at time t = 15 and once he gets to an ATM machine he will spend exactly 5 time units at the machine and then depart.

Our simulation begins by reading arrivals from the file and creating entries in a list where each entry represents a person and each entry represents an arrival event (type 'A') which has an arrival time and a service time. That event list will be sorted on the time that the even will happen which is the arrival time.

What about departure events (type 'D')? When a person gets to an ATM machine we will create a departure event and put that event in the event list. The time when that departure event will happen is the current time (wallClockTime) plus the service time—the time the person will spend at the ATM. Now that we know the departure time, we create a new event of type 'D' and the time that event will happen will be the departure time we just calculated (wallClockTime plus service time). We put that new event into the event list and sort the list.

There are a couple of other details. If a person arrives (event type 'A') and finds that the line is empty and there is an ATM available, they will, of course, go directly to an ATM. If a person departs (event type 'D') from an ATM machine and there is someone in line, we will move that person to an ATM machine (as above, we will calculate the time when they will finish and depart, and, using that calculated time, add a new departure event (type 'D') to the event list).

Here is the event-driven algorithm:

```
# read the file of arrival events and put the events into the event list, then sort the list
# while the event list is not empty:
#   pop the next event off the event list
#   wallClockTime = the time of the event
#   if event type == 'A':  # person arrives at the line
#      if the line is empty and an ATM is available:
#         send the person to an available ATM machine which means: create a new 'D' event with an
#             event time calculated from the wallClockTime plus the person's service time
#         add this new 'D' event to the event list
#      else:  # put the person in line because there is nowhere else to go
#         add the person to the back of the line
#   else:  # event is type == 'D' someone is leaving an ATM machine
#      if the line is empty:
#         register the ATM machine as empty and available
#      else:  # there are people in line and this ATM just became available
#         get the ATM number from the 'D' event information
#         pop the person (event) from the line
#         send the person to the available ATM machine which means: create a new 'D' event with an
#             event time calculated from the wallClockTime plus the person's service time
#         add this new 'D' event to the event list
#   print the event, the line and ATM's in use
```

As complicated as that algorithm seems, it is almost precisely matches line-for-line the actual Python code.

Data Structures

It works really well to have a class Event and a class EventList. The EventList will be a list of Events.

The Event class will need this data:

- eventTime: time when this event happens, either arrival time or departure time, depending on the type
- serviceTime: time spent at an ATM
- eventType: 'A' or 'D'
- ATMnumber: if eventType 'D', this is the ATM that the person is at
- ~~lineArrivalTime: if type 'A' and the person is in line, this keeps the time of when the person arrived at the back of the line. We need this data so we can keep track of how long this person was in line.~~ It is optional because you can use "eventTime".

I also had those items as parameters with default values so I didn't have to specify the ones I didn't need.
The Event class also needs to overload the compare operator so Events can be sorted. If you have this operator in the Event class, the sort will work.

```
def __cmp__(self,other):
    return cmp(self.eventTime, other.eventTime)
```

A __str__ method is very useful for printing Events.
~~A *setArrivalTime* method is very useful, too, if you use the optional "lineArrivalTime".~~
Of course, it needs an __init__ method.

The EventList class will need:

- eList: a list of events

The EventList needs a method to *insert* an event into the EventList, and a *pop* method to pop an Event from the EventList and return the event.
An *empty* method is useful – it returns true if the EventList is empty.
A __str__ method is useful for printing an EventList.
Of course, you need an __init__ method.

The data structure for the line can simply be a list. A list of what? Well, it is easiest to simply use Event objects since you already have that class and they have information you need in the line (such as serviceTime and lineArrivalTime once you set it).

I also found it easiest to keep a list of available ATM machines. There are four ATM machines in the International Center so a simple list initialized as [1,2,3,4] representing the four machines worked fine.

Time in Line

How do we keep track of time in line? The time in line is the time from when we arrive at the back of the line until we go to an ATM. Using the lineArrivalTime data in an Event, we can set it to the wallClockTime when we arrive at the back of the line, and then when we leave the line to go to an ATM machine we can simply calculate the time in line as the wallClockTime – lineArrivalTime.

To calculate the average time in line we need to add up the total time in line spent by everyone and divide by how many people were in line.

Note that, if someone arrived to find an empty line and an available ATM machine, their time in line is zero.

Output

After each event:

- Output the event that happened. For each event output the eventTime, the serviceTime, and the event Type.
- Output the line of people waiting for ATMs: since we are recording each "person" as an Event, this will be a list of events in line for ATMs. For each event in line, output its arrival time and its service time.
- Output each ATM: output the ATM number (1,2,3, or 4) and the service time

At the end of the program run:

- Output the average time spent by people in each line.

Deliverables

You must use handin to turn in the file **proj10.py** – this is your source code solution; be sure to include your section, the date, the project number and comments describing your code. Please be sure to use the specified file name, and save a copy of your proj10.py file to your H drive as a backup.

Other good information

Notes:

1. Extra Credit: Program this project for one line If your are attempting extra credit, once that is working you can create four lines—simply four lists (one for each line) and a few other minor changes. Run the simulation twice (cut-and-paste or, better, create a function): run the simulation with one line and then with four lines—at the end, compare the average time in line.