CSC148 Fall 2018 Home Forum MarkUs Lecture Notes Preps Assignments Syllabus Software Exams Course Team Anonymous Feedback

Assignment 1: Elevator Simulations

Check the updated PythonTA tweaks section below!

Sample tests now posted!

One of the oldest and most widespread uses of computer programming is to create and run *simulations*, which are programs that model a particular domain, be it a complex chemical or physical system, the spread of a contagious disease, or financial systems like the stock market.

One of most useful features of a computer simulation is that they are *parameterizable*, meaning we can run the same simulation program several times while varying different input parameters to see how the system's behaviour changes.

On this assignment, you'll produce a simulation of a (slightly odd) elevator system, in which elevators carry people between different floors of a building, using different algorithms for deciding how elevators move between floors. We'll use a *time-based* simulation, for which the state of the simulation is updated in rounds, where each round represents a fixed length of time (e.g., 1 second). By varying the different parameters of the system, you'll be able to compare different algorithms, making judgments about whether one is better than another in some cases, or in all cases.

Reminder: general assignment guidelines

Please review the General Assignment Guidelines carefully before starting this assignment.

Problem domain overview

The context of our elevator simulation is a building which has a specified number of floors (numbered starting at Floor 1) and elevators that can move between each floor. People can arrive at the elevators at any floor, and elevators move to pick up people and (hopefully) take them to their desired destination floor.

Our elevator simulation consists of three main parts:

1. The basic entities contained in the simulation (people and elevators).

2. The update algorithms that determine how the entities change over time (how new people appear in the simulation, how elevators decide to move across floors).

3. The main runner of the simulation, which is responsible for setting up the initial parameters of the simulation, running the simulation, determining when the simulation should end, and reporting the results of the simulation.

In this section, we'll describe the expected behaviour for all three of these parts.

People

Every round of the simulation, zero or more new people arrive at the elevators. A person in the simulation has three characteristics: which floor they started on, which floor they want to go to, and the number of *simulation rounds* they have been waiting to reach their target floor. The building's floors are numbered starting at 1, meaning each person's starting and target floor should be between 1 and the maximum floor number of the building, inclusive.

Each person's waiting time increases both when they're waiting at a floor, and when they're traveling on an elevator.

Elevators

The simulation has a fixed number of elevators that all begin the simulation at Floor 1 of the building. An elevator can move one floor (up or down) per simulation round, and so we only need to track which floor each elevator is on, and *don't* need to worry about an elevator being "between floors". Each elevator keeps track of its passengers (which people are currently on it), as well as its maximum capacity—if an elevator is full, no more people can board it. Each elevator must be able to track the order in which people boarded it.

Arrival generation algorithms

At the start of each simulation round, new people may arrive at the elevators. There are different ways to decide how many people arrive at a given round, and the starting and target floors of each person. On this assignment, you'll implement two different approaches described below.

Random generation

Each round, a fixed number of people are randomly generated; their starting and target floors are all random, with the requirement that a person's starting and target floor can't be the same.

Generation from file data

A **csv file** is a text file that uses commas to separate pieces of data. ("csv" stands for "comma-separated values")

Our second approach for arrival generation is to read arrivals from a .csv file, in the following format:

- Each line of the file represents all of the arrivals for a certain round number.

- On a single line, the first value is the *round number* that this line specifies. This is followed by an even number of other entries, divided into pairs of numbers. In each pair, the first number is the starting floor and the second number is the target floor of a new arrival. The arrivals occur in the order the pairs appear in the line. Each line must have at least one pair (i.e., store at least one new arrival).

For example, the following data file

```
1, 1, 4, 5, 3
```

```
3, 1, 2
```

```
5, 4, 2
```

represents the following arrivals:

- Round 1: one person whose starting floor is 1 and target floor is 4, and another person whose starting floor is 5 and target floor is 3, in that order.
- Round 3: one person whose starting floor is 1 and target floor is 2.
- Round 5: one person whose starting floor is 4 and target floor is 2.

You may make the following assumptions about input files for this assignment:

1. They are all in the format described above.
2. The round numbers are non-negative, and are less than the maximum number of rounds in the simulation. (Note: round numbers start at *zero*, not one.)
3. Each round number is unique (no two lines start with the same number). But *don't* assume that the lines are in any particular order.
4. Each person has starting and target floors that are positive, and do not exceed the maximum number of floors in the simulation.

5. Each person has a target floor that's different from their starting floor.

Elevator moving algorithms

Each round, an *elevator moving algorithm* makes a decision about where each elevator should move. Because an elevator can only move one floor per round, this decision can have one of three outputs: the elevator should move up, move down, or stay in the same location.

A *moving algorithm* receives as input two values: a list of all the elevators in the simulation, and a dictionary mapping floor numbers to a list of people who are waiting on that floor. It outputs a list of decisions (one for each elevator) specifying in which direction it should move.

This is an extremely flexible model of how elevators move (in real-life, the use of elevator buttons makes this much more constrained), and the reason we do this is so that that you can implement a variety of fun and interesting elevator algorithms! On this assignment, you will implement the following three algorithms:

Random algorithm

The algorithm makes a random decision for each elevator, choosing between each of the three possibilities with equal probability. These choices are made independently for each elevator.

Pushy Passenger algorithm

This algorithm makes a decision independently for each elevator.

If the elevator is empty (has no passengers), it moves towards the *lowest* floor that has at least one person waiting, or stays still if there are no people waiting. Because the decisions are independent for each elevator, if at least one person is waiting, *all* empty elevators move to the same floor.

If the elevator isn't empty, it moves towards the target floor of the *first* passenger who boarded the elevator.

Short-sighted algorithm

This algorithm makes a decision independently for each elevator.

If the elevator is empty, it moves towards the *closest* floor that has at least one person waiting, or stays still if there are no people waiting. In the case of ties (e.g. if the elevator is at floor 3, and there are people waiting at floors 2 and 4), it moves towards the *lower* floor. As in the previous algorithm, because the decisions are independent for each elevator, *all* empty elevators move to the same floor. (*Updated*: the previous sentence didn't make sense for this algorithm, and should be ignored.)

If the elevator isn't empty, it moves towards the closest target floor of all passengers who are on the elevator, again breaking ties by moving towards the *lower* floor. In this case, the order in which people boarded does not matter.

Simulation

The main simulation program itself is divided into three stages: initializing the simulation with a given configuration, running the simulation, and then calculating and reporting statistics at the end of the simulation.

Initializing and configuring the simulation

The simulation's configuration is set through a dictionary with the following keys:

- `num_floors`: the number of floors in the building (the floors are numbered starting from 1, to the highest floor is numbered `num_floors`)
- `num_elevators`: the number of elevators
- `elevator_capacity`: the capacity of each elevator (every elevator has the same capacity)
- `num_people_per_round`: the number of of people per round to spawn, or `None` if this is left up to the algorithm
- `arrival_generator`: the algorithm to use to generate new arrivals for the simulation
- `moving_algorithm`: the algorithm to use to determine how elevators move in this simulation
- `visualize`: whether to visualize the simulation using Pygame

Simulation restrictions

You are allowed to make the following *assumptions* about the simulation configuration (and don't need to check for them!):

- All values are of the correct type (the first four listed above are all non-negative integers, the next two are objects representing the algorithms, and the last is a boolean)
- `num_floors >= 2`
- `num_elevators >= 1`
- `elevator_capacity >= 1`

Simulation rounds

The overall simulation proceeds in rounds (each round representing one fixed length of time, which for us is the time it takes for an elevator to move from one floor to another and for people to get on or off the elevators). Each round consists of *four* different phases, which are described here.

1. *New arrivals.* Using an arrival generator algorithm, new people are added to the simulation.
2. *Elevator disembarking.* Everyone who is on an elevator and at their target floor leaves

the elevator.

3. *Elevator loading.* People board elevators on their floor that have space, according to the following restrictions:

    - People board in order of when they arrived at the floor (first arrival boards first).

    - If there are multiple elevators with space at the same floor, the elevators are boarded in left-to-right order. (The simulation should store the elevators in a list, where the front of the list represents the leftmost elevator.)

For example, if there are ten people waiting and two elevators with three empty spots each, then the first three people board the leftmost elevator, the next three people board the second elevator, and the remaining four people don't board any elevator.

Note: people board elevators without knowing what direction the elevator is traveling in. This means that it's possible a person will board an elevator, only to be taken away from their target floor! That's life in this simulation.

4. *Elevators move.* Elevators move based on the decisions made by the simulation's elevator moving algorithm.

Reporting statistics

At the end of the simulation, you are responsible for reporting summary statistics for the simulation, in a dictionary with the following keys:

- `num_iterations`: the number of simulation rounds that took place

- `total_people`: the number of people that arrived at some point during the simulation

- `people_completed`: the number of people who reached their target destination by the end of the simulation

- `max_time`: the maximum time someone spent before reaching their target floor during the simulation (note that this includes time spent waiting on a floor and travelling on an elevator)

- `min_time`: the minimum time someone spent before reaching their target floor

- `avg_time`: the average time someone spent before reaching their target floor, *rounded down* to the nearest integer

For `max_time`, `min_time`, and `avg_time`, these three statistics should *only* take into account the people who actually reached their target floor. Report **-1** if no person reached their target floor during the simulation run.

Visualizing the simulation

We have provided a simple visualization library across two files (`visualizer.py` and `sprites.py`) for visualizing the actions in a simulation. You are not responsible for implementing anything in these files. Instead, you should *read their documentation* to understand how to use them correctly, so that you are able to call the appropriate methods to visualize all aspects of your simulation, and inherit from the two main sprite abstract classes. (A

sprite is an image that represents an entity in a graphical display.)

Note: due to the limitations of our visualizer, the simulation looks best when given fewer than eight-ish floors and elevators. However, your program should be able to handle simulation configurations beyond this range! When testing these cases, set `visualize` to `False` so that the simulation simply runs in the background, without visualizing anything.

Starter code

Download the following files and put them into your `assignments/a1` folder.

- `algorithms.py`
- `entities.py`
- `simulation.py`
- `sprites.py`
- `visualizer.py` (updated Oct 9 for minor cosmetic tweak; you can choose to update your file, or safely ignore)
- `sample_arrivals.csv`

Also, download this people.tar file, and extract its contents into your a1 folder to create a subfolder called people. Inside that folder, you should find five different images.

Closer to the deadline, we'll post some sample test files to give you a sense of some of the testing we'll be doing—but don't wait until then to start testing your code!

Here are some sample tests. Note that these are *extremely* incomplete, and are meant to give you a sense of the kind of testing we'll do, not to thoroughly assess the correctness of your work!

- a1_sample_test.py

(Updated) PythonTA tweaks

We have encountered one error and a few unforeseen limitations when using python_ta, and so please do the following:

1. *Update your version of* `python_ta`.

    a. In PyCharm, open up the Settings/Preferences window, and go to Project: csc148 -> Project Interpreter.

    b. On the right-hand side, click on the + icon, and search for `python-ta`, and select it.

c. In the bottom-right corner, select the “Specify version” checkbox, and make sure **1.4.1** is selected.

d. Click on “Install Package”.

2. In the three code files you’re working with, add the following key-value pairs to the `config` dictionary when you call `python_ta.check_all`:

   a. `'max-attributes': 12`

   b. `'disable': ['R0201']` (we included this already in `algorithms.py`, but you can add it to the other files)

These options relax some of the checks that `python_ta` is performing, and we’ll use these options when grading your work.

Task 1: Setting up the simulation and elevators

Your first task is to complete the `Simulation.__init__` method in `simulation.py`, so that all of its attributes are initialized. This should include constructing the necessary *elevators* for this simulation; we have provided the stub of a class `Elevator` in `entities.py` that you should complete for this task, according to the description given in the *Elevators* section above.

Please read through the documentation provided in both of those files, as they will provide some detail and further instructions.

Check your work

After completing this, you should be able to run the provided `sample_run` function and see Pygame show a window with the number of elevators and floors it has configured.

Tip: the Pygame window should close itself after 15 rounds, but if you ever want to stop the simulation early, simply go to PyCharm's Run pane and click on the red Square ("stop") button.

Task 2: People and arrivals

Next, complete the design and implementation of `Person` in `entities.py` again based on the description given in the *People* section above.

Then, read through the abstract class `ArrivalGenerator` in `algorithms.py`, and implement the two subclasses corresponding to the algorithms described in *Arrival generation algorithms* above. We've provided a sample `arrivals.csv` that is compatible with the configuration given in the starter code, but we encourage you to create your own files as well!

Lastly, implement `Simulation._generate_arrivals` to actually run your arrival generation algorithms. This should both update the `waiting` instance attribute of the simulation, as well as call the `Visualizer.show_arrivals` method appropriately to actually show the new arrivals in the Pygame window.

Check your work

Try running the simulation on both kinds of arrival generation algorithms. You should see people arriving on different floors of the building.

Task 3: Elevator boarding

Now that you have people and elevators, you can complete the second stage of the simulation round: having people board elevators. This should be relatively straightforward after the previous two steps, and can be done entirely in `Simulation._handle_boarding` with one or two helper functions to help split up your code. You'll need to remember to update the state of each elevator as it takes on new passengers, and to visualize the change by calling `Visualizer.show_boarding` appropriately.

Check your work

Now when you run the sample simulation, you should see that in addition to people arriving at floors, anyone who arrives on the first floor moves to the leftmost elevator that still has room. Play around with the arrival generation so that you can generate a lot of arrivals at floor 1, to check that you correctly handle the case when some elevators are full.

If you properly read through all the documentation in `sprites.py`, you should see some visual effects as the simulation runs: elevators getting full, and people getting angrier. **Note**: you'll notice that the visualizer seems to draw sprites in a random (or reverse) order, so the "oldest" sprites are covered by newer ones. This is normal, and not a bug with your code!

Task 4: Random elevator movements and elevator disembarking

Next, read through the abstract `MovingAlgorithm` class to understand how we expect you to represent *elevator moving algorithms* in your program. Then, implement the `RandomAlgorithm` subclass (this one's easiest of the three algorithms to implement), and then complete `Simulation._move_elevators` to use your algorithm to move each elevator. As usual, you'll need to both update the state of the elevators, and visualize the change by calling `Visualizer.show_elevator_moves` appropriately.

Then to complete a full simulation round, implement `Simulation._handle_leaving` so that when an elevator moves to the target floor of one of its passengers, that passenger leaves the elevator. You'll need to call `Visualizer.show_disembarking` appropriately to visualize this happening.

Check your work

After completing these two steps, you should now have a fully working simulation that executes (and visualizes!) all four stages of a simulation round: new people arriving, people disembarking from elevators once they've reached their target floor, people boarding elevators that have space, and elevators moving according to your random algorithm.

Try varying the configurations of the sample simulation runs: you'll likely find some bugs in at least one of these stages, and now is a good time to fix them!

Task 5: Tracking and reporting statistics

Review the simulation statistics you are required to track for a simulation run. Your task here is to add to your existing code so that you can keep track of and report these statistics. Don't be afraid to make fairly significant code changes here (making sure you don't change any of the starter code's *public* interfaces, of course). This might require extra attributes and/or methods in some of your classes; this is certainly allowed, just make sure you document them properly.

Check your work

Because you're using a random algorithm to move elevators, you won't be able to check the exact value of all of the statistics you calculate, but you should be able to check that you're computing all the statistics we require for this assignment, and that the values for these statistics lie in a reasonable range.

Task 6: Implementing the two non-random elevator algorithms

Your last remaining task is to implement the other two elevator moving algorithms, according to the descriptions given above. After this, you should be able to simply pass in instances of the other two moving algorithms in the simulation configuration, and run the simulation without changing any other code!

This is the beauty of programming to a *shared public interface* like `MovingAlgorithm`.

Check your work

At this point, it may be tempting to just run your simulation a bunch of times and watch the Pygame window, but this is not a very robust approach to checking whether your algorithms are correct. Instead, we recommend using your `FileArrivals` class to create *test inputs* for the arrivals, and then carefully trace what should actually happen on small inputs, and check whether your algorithms behave as intended. You should be able to predict exactly where each elevator will move every round, and what the final simulation statistics will be!

Polish!

Take some time to polish up. This step will improve your mark, but it also feels so good. Here are some things you can do:

- Pay attention to any violations of the Python style guidelines that PyCharm points out. Fix them!

- In each module, run the provided `python_ta.check_all()` code to check for errors. Fix them!

- Check your docstrings to make sure they are precise and complete and that they follow the conventions of the Function Design Recipe and the Class Design Recipe.

- Remove any code you added just for debugging, such as `print` function calls.

- Remove any `pass` statement where you have added the necessary code.

- Remove the word "TODO" wherever you have completed the task.

- Take pride in your gorgeous code! This assignment is a significant piece of software, and you should be proud of the work you've done.

Submission instructions

1. Login to MarkUs and create a group for the assignment, or specify that you're working alone.

2. **DOES YOUR CODE RUN**?!

3. Submit the files `algorithms.py`, `entities.py`, and `simulation.py`. Don't submit any other files.

4. On a Teaching Lab machine, download all of the files you submitted into a brand-new folder, together with the other starter code files provided here. Test your code thoroughly. *Your code will be tested on the Teaching Lab machines, so it must run in that environment.*

5. Congratulations, you are finished with your first assignment in CSC148! Go have some chocolate or do a cartwheel. :)

For course-related questions, please contact **csc148-2018-09 at cs.toronto.edu**.