CSC110 Tutorial 11: Object-Oriented Design and Discrete-Event Simulations

Print this handout

This week, we engaged in an extended case study of building a simulation of a food delivery system. Along the way, you learned about how to break down a complex system into multiple classes (a mixture of data classes and general Python classes) and the design of a *discrete-event simulation*.

In this tutorial, you will reinforce and extend your understanding of these concepts. In Exercises 1 and 2, you'll improve our basic simulation to make it more realistic by incorporating entity locations and using randomly-generated menus when our customers actually order some food! In Exercise 3, you'll conduct one last timing experiment to compare two different priority queue implementations. And in Exercise 4, you'll complete an open-ended design activity, looking at different ways of extending your simulation based on new requirements.

Finally, a reminder that this is your **last tutorial session for CSC110**! If you've been coming to tutorials regularly, please be sure to thank your TA for all the work they put in this semester to help you with your learning. 🙏

Starter files

All exercises in this tutorial share the same starter files, which are a slightly updated version of all the code we covered this week. Please download `tutorial11.zip` and extract all the files into your tutorial folder for this week.

We recommend taking a few minutes to open each starter file to remind yourself of the different parts of the code that we covered in lecture this week.

Exercise 1: Incorporating Distances

Recall that we defined a `NewOrderEvent` class to represent an order being placed in our food delivery system. Our current implementation of `NewOrderEvent.handle_event` is a bit unsatisfying because it computes a fixed time (10 minutes) for an order to complete. In reality, we'd expect the time for the delivery to depend on factors like the locations of the courier, vendor, and customer involved in the delivery (as well as other factors like time of day, traffic levels, etc.).

So your first task in this tutorial is to improve our simulation by doing the following:

1. In `food_delivery_system.py`, implement the *top-level functions* `calculate_distance` and `estimate_delivery_time`. For `calculate_distance`, you'll implement one last mathematical formula for this course:

   > Given two (latitude, longitude) points (ϕ_1, λ_1) and (ϕ_2, λ_2) on a sphere of radius r, the distance between these two points on the sphere (also called *spherical distance*) is calculated as follows:
   >
   > 1. Calculate differences $\Delta\phi = \phi_2 - \phi_1$ and $\Delta\lambda = \lambda_2 - \lambda_1$, *in radians*.
   >   - *Hint*: you can use the `math.radians` function to convert from degrees to radians.
   > 2. Calculate the central angle between the points:
   >
   > $$\theta = 2 \cdot \sin^{-1}\left(\sqrt{\sin^2\left(\frac{\Delta\phi}{2}\right) + \cos\phi_1 \cos\phi_2 \sin^2\left(\frac{\Delta\lambda}{2}\right)}\right)$$
   >
   > 3. The spherical distance is $\theta \cdot r$.

2. Modify the `FoodDeliverySystem.place_order` method so that its return type is now `Optional[int]` instead of `bool`, so that:
   - If no courier is assigned, return `None`.
   - If a courier is assigned, return the estimated delivery time (an `int`, in seconds).
3. Finally, in `NewOrderEvent.handle_event`, use the new return value to set the timestamp of the corresponding `CompleteOrderEvent`.

Since we haven't changed the overall simulation structure, running the simulation should be about the same. But now the orders that cover longer distances should generally take longer.

Exercise 2: Generating Menus

Open `simulation.py`, and review the `run_example` function, which runs an example simulation. One current limitation is the lack of menu data! All of our vendors have very simplistic menus, and all of our orders also have empty `food_items`. Let's fix that now.

1. Recall that a menu is a `dict[str, float]`, where each key the name of a dish and each corresponding value is the price of that dish.

   In `simulation.py`, implement the functions `load_menu_data` (which reads possible menu items from a file we've provided) and `generate_random_menu`, which uses this data to return a random menu.
2. Modify `Simulation._generate_system` to generate a random menu for each vendor that's created.
3. In `GenerateOrdersEvent.handle_event`, modify how each order is created so that it has a random `food_items` dictionary based on the chosen vendor's menu.

   You can use a similar approach to `generate_random_menu`, except that the returned dictionary is a `dict[str, int]`: each key's corresponding value is the *quantity* of that food item in the order.
4. With just these changes alone, you won't see any changes in the direct output of your simulation. Try modifying the `NewOrderEvent.__str__` method to display the `food_items` in each order when the event is printed, and then re-run your simulation.

   (Make sure the `print(event)` line in `run_simulation` is uncommented!)

Exercise 3: Multiple `EventQueue` implementations

Our discrete-event simulation is based on the Priority Queue abstract data type we learned about last week. The various list-based implementations of this ADT that we covered in lecture and the course notes have a worst-case running time of $\Theta(n)$ for at least one of their `enqueue`/`dequeue` implementations.

It turns out that Python has a built-in module `heapq` which implements a Priority Queue using a *heap* data structure, which you'll learn about in CSC263/265.[1] In `event_queue.py`, we've implemented a separate `EventQueue` subclass called `EventQueueHeap` that uses this module to implement the `enqueue` and `dequeue` methods with a worst-case running time of $\Theta(\log n)$.

In this exercise, you'll write one final timing experiment to compare the time taken to run our simulation using the two different `EventQueue` implementations.

1. Try running your simulation over the span of 100 days, and using `timeit.timeit` to measure the time taken to run your simulation.
   - Tip: review Tutorial 8, Exercise 3 to remind yourself how you use the `timeit` module.
2. Then in `simulation.py`, make the `events` variable in `run_simulation` be an `EventQueueHeap` instead of an `EventQueueList`. You'll need to add the `EventQueueHeap` class to an import statement at the top of this file.
3. Re-run your simulation, again using `timeit.timeit` to measure the time taken.
4. Repeat for a longer duration for `GenerateOrdersEvent` (or modify its `handle_event` method to make the `timedelta` a few seconds rather than a few minutes). You should see that the running time for the `EventQueueList` grows a lot faster than the running time for `EventQueueHeap`!

Exercise 4: Augmenting our class design

Class design is a *iterative* process, meaning it is something we return to and change over time. Sometimes we want to support new functionality, or remove or simplify other functionality. Sometimes our software requirements change because our clients or users decide they want something different!

This week's design of the food delivery simulation was fairly static—for the purposes of lecture, we told you what our design would be, and we focused on understanding and implementing it. In Exercises 1 and 2, we gave you explicit instructions on how to modify our design to implement new features in our simulation. In this exercise, you'll get practice doing an open-ended design of a few feature yourself, without us telling you exactly what to do!

> **New requirement**. Mario looks at the food delivery simulation we've put together and says:
>
> > That's all fine, but I want to understand *costs* as well. How much commission should we charge per order? Should we use a flat fee (e.g., $5/order), or a percentage of the order total, or both? How much do we pay our couriers per order? A flat fee, percentage of total order, rate based on distance travelled? A combination of these?
> >
> > How will these decisions affect the earnings of each courier and vendor? How will it affect our earnings?

There's a lot to unpack there. Before writing any code, let's work to break down what Mario is asking for and think about how this impacts our existing design.

1. Consider our four basic entity data classes, `Vendor`, `Customer`, `Courier`, and `Order`.

   Are there any new *instance attributes* that would make sense to store with these classes? Would these attributes have *default values* or would we need to specify what they are when we create new instances of these data classes?
2. We should keep track of the amount of money earned through commission when completing different orders. This makes sense to store as an instance attribute. What class should we make this instance attribute belong to? Is there more than one possible option?
3. We'll need to modify our existing code to compute costs of each order and the amount paid to couriers, vendors, and to the company through commission.

   Where in our program should these costs be calculated in our code? Is there more than one possible option?
4. Mario mentioned several possible scenarios for calculating commission earned by the company and payments to couriers. How could we change the design of our simulation to allow a user to specify these different scenarios, and what would we need to be able to keep track of these options?

After brainstorming answers for each of these questions, we *strongly* recommend comparing notes with your classmates and your TA.

You probably will not have time to do any implementation of your work before the end of the tutorial, and that's okay! This exercise was mainly to get you thinking about the kinds of design decisions that you'd need to make, and less about the actual programming.

For extra practice after this tutorial is over, try implementing your decisions. Your changes should affect a few different classes, and will be larger than your work on Exercise 1 or 2 for this tutorial.

---

1. Remarkably, this implementation also uses a plain Python list, though it does so in a very different way than how `EventQueueList` works. ↩