Hopefully, you'll find these helpful!

**Figuring out which Server has the next scheduled event**

**nes: Server = [srvr for srvr in sorted(self.\_servers.values(),  
 key=lambda srvr: srvr.getNextEventTime())][0]**

This code 1) sorts the servers based on their next event time, so that the server with the earliest next event is in the zero position of the sorted list and then 2) extracts the zero server (i.e. the one with the next event). From their, you can ask the server for its nextEventTime and, if you need it, nextEventType.

The use of *sorted* here bears a little explanation. Normally, sorted will simply try to sort the objects. This works fine for basic types such as ints and strings, but doesn't work very well on objects. Here we use the key= option of sorted to specify a transformed value to use for the sort key. In this case, we use a lambda expression to get the next event time from the server, which becomes the sortkey.

**Identify available servers**

**return {**[**srvr.id**](http://srvr.id/)**: srvr for srvr in self.\_servers.values() if srvr.isAvailable}**

I found it useful to be able to figure out which Servers were available. So the code above returns a dictionary (key = serverId, value = Server instance) that contains only the available servers. This code involves a comprehension with a condition. When generating the comprehension (in this case, once for every server for the queue), Python will include that server only if the condition is satisifed (srvr.isAvailable in this case). List, dictionary, and generator expressions all work similarly (w.r.t. to the for and if clauses).

**Get the number of available servers**

**return sum([srvr.isAvailable for srvr in self.\_servers.values()])**

Again, I found it useful to get a count of available servers (since when advancing customers to service, the maximum number of customers that can enter service is the number of available servers). In this code, the list comprehension contains a True or False for every server. Since Python sees True as 1 and False as 0, summing that set of values represents the number of available servers.

**Making Simulation an iterable:**

**for stage in self.\_stages.values():  
 # print(f'Stage: {**[**stage.id**](http://stage.id/)**}, {type(stage)}')  
 if isinstance(stage, SystemExit):  
 for cust in stage:  
 yield cust**

Because a simulation may have multiple system exits, making Simulation an iterable means providing an \_\_iter\_\_ method that returns customers from ALL system exits. Thus making Simulation an iterable is a bit more challenging than just returning a generator expression which will work for SystemExit (and we discussed this technique in class).

The code above can be defined in the \_\_iter\_\_ function. Any time you use yield (which we also briefly discussed):

1. Python will return the specified argument (e.g. cust in this case)
2. Retain the execution state of the method
3. Resume at the next point of execution the next time the method is called.

The long and short of this is that using *yield* creates a *generator function* which works the same way an iterable does. Thus, in the code above, you can see that I loop over all of the stages, because the Simulation doesn't know which stages are SystemExits. If a stage is a SystemExit, then the code returns its customers one at a time (using the yield) until they have all been returned. Then, the loop continues with the next SimulationStage until there are no more. The end result is that all customers from all SystemExits are returned.

To *watch* this execute, you can uncomment the print statement.