Using the starter `ssq.py` that we developed in class, complete an event-driven simulation implementation of a single-server queue in Python using the `simulus` library.

Some things to keep in mind as you work on your implementation:

- You will need two different functions, one each to correspond to the event types of arrival and completion-of-service. In each of those functions, handle the corresponding algorithmic details for that event type (whether an arrival to the system or a completion of service).
- Use simulus's `sim.sched` whenever you need to schedule another event occurrence (where `sim` is defined as a global-scope object of `simulus.simulator()`). The details of how `sim.sched` can schedule and call your event-type functions are provided in the videos posted on Lyceum.
- Use the separate `QueueStats` class, and an object passed to your functions, to store and update statistics that you will need to compute appropriate time-averaged statistics for the model. In particular, you will need to track:
  - current number in the system (your state variable)
  - total number of arrivals
  - total number of departures
  - the sum-of-rectangles area for the number-in-the-system skyline function
  - the sum-of-rectangles area for the number-in-the-queue skyline function
  - the sum-of-rectangles area for the number-in-the-server skyline function (used to compute utilization)
  - the time of the last event to have occurred in simulated time, for use in computing the area of a rectangle (see above) relative to `sim.now`
- You will need to update your area statistics at the very start of your arrival and completion-of-service functions, before you update the number in the system. **NOTE:** Make sure to carefully think about the use of the number in the system with respect to adding to the areas for number-in-queue and number-in-service.
- Print some useful statistics at the end of simulation, similar to that shown below:

```
# Arrivals:        10000
# Completions:     10000
# in system @ end: 0
TA # in system:    8.09491
TA # in queue:     7.19845
utilization:       0.89645
```

- You can, and should, compare results of your own simulation relative to that produced by `ssq` from `simEd` in `R`. Specifically, you can benchmark your Python implementation by comparing to some meaningful statistics from the R implementation, e.g.,:

```
> output <- ssq(maxArrivals = 10000, showOutput=FALSE)
|======================================================================| 100%
> output$avgNumInSystem
[1] 10.65661
> output$avgNumInQueue
[1] 9.748865
> output$utilization
[1] 0.9077439
```

- To make sure that you can process an exact number of arrivals using `simulus`, don't provide a maximum time (i.e., just call `sim.run()`). However, to ensure that the simulation will eventually stop, you will need to include some appropriate "close-the-door" logic in your arrival function. That is, only schedule a new arrival if the cumulative number of arrivals is less than some globally-defined maximum number of arrivals (e.g., 10 000 in the example above).

Experimentation: At the bottom of your Python file, include block comments providing evidence that output from your Python implementation is consistent with that from `simEd`'s `ssq`. Specifically, provide output from at least five different runs of `ssq` from `simEd`, and compare that with the output from at least five different runs of your Python implementation.

Note: You can use `random.seed` prior to constructing `simulus.simulator()`, but do not expect the output using a given seed in your Python implementation to be identical to that using the same seed in R — the underlying RNG implementations are different in R versus Python.