SimPy Part 1

SimPy for Discrete Event Simulation Part 1

Dr Daniel Chalk

Quick Recap

Today, we’re going to show you in more detail how to build a Discrete Event Simulation model using SimPy.

But first, let’s recap the main points from the introductory session.

Components of Discrete Event Simulation

Entities are the things flowing through the sequential processes in the model (e.g. patients, telephone calls, blood test results)

Generators are the way in which entities enter the model and come into being (e.g. brought in by paramedics, self-presenting at the ED)

Inter-arrival Times specify the time between entities being generated (arriving in the model)

Components of Discrete Event Simulation

Activities / Servers represent the activities that happen to entities (e.g. triage, treatment, ward admission)

Activity / Server Time represents the amount of time it takes for an activity to happen to an entity.

Resources are required for activities to take place and may be shared between activities (e.g. nurse, doctor, receptionist, bed)

Queues are where entities are held until an activity has capacity and the required resources to begin.

Sinks are how entities leave the model.

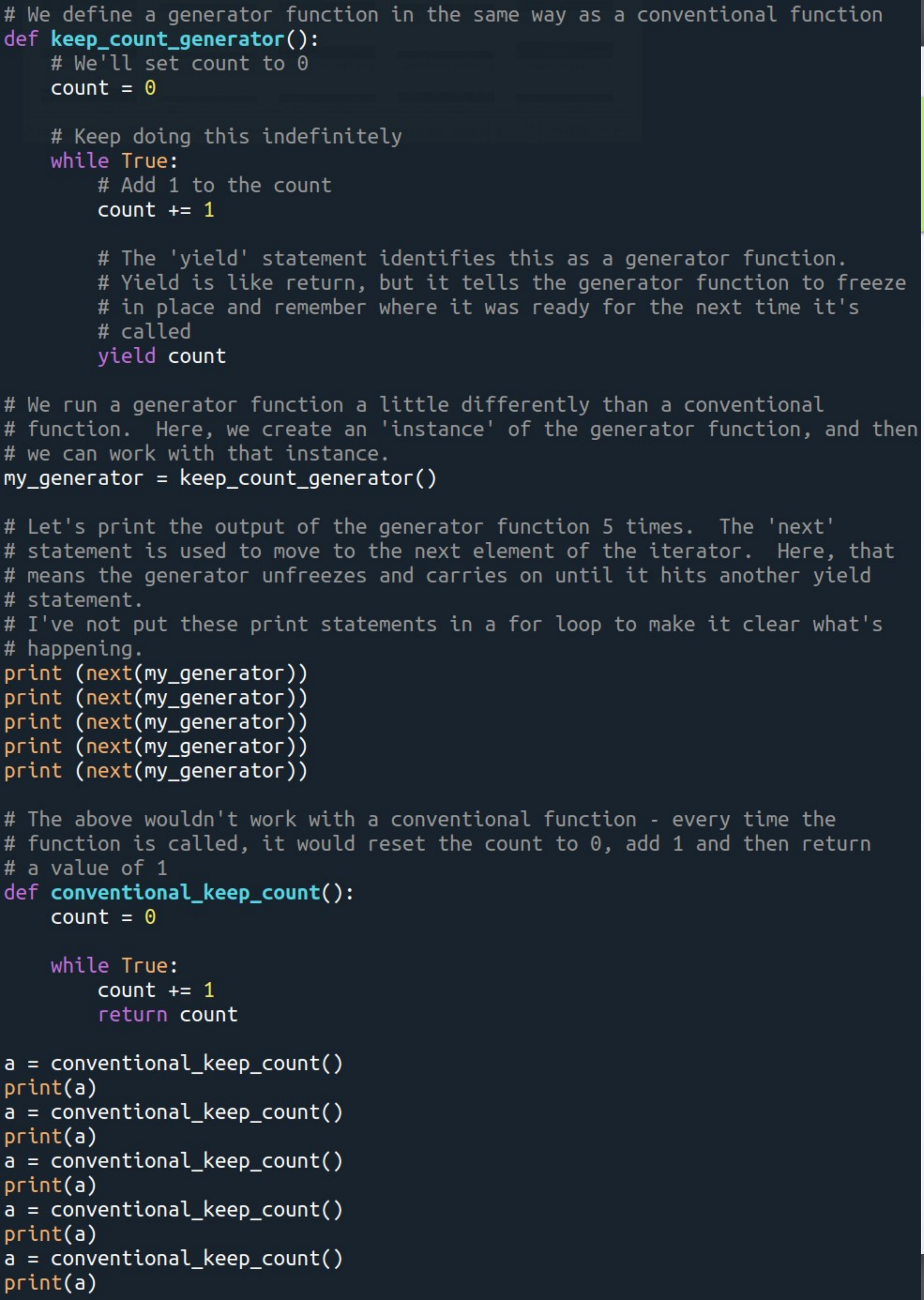
Generator Functions

Conventional functions in Python are called, and then run with some (optional) inputs, and then finish (usually by returning some output).

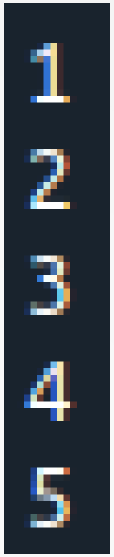
Generator functions remember where they were and what they did when control is passed back (they retain their “local state”), so that they can continue where they left off, and can be used as powerful *iterators* (for and while loops are other examples of *iterators*).

This is *very* useful where we want state to be maintained (e.g. during a simulation run)

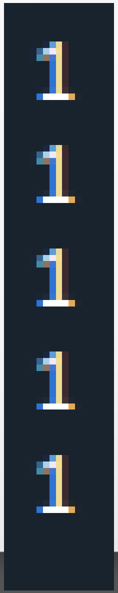
Let’s look at a very simple example of a generator function to see how they work (simpy_1.py).



```python
# We define a generator function in the same way as a conventional function
def keep_count_generator():
    # We'll set count to 0
    count = 0

    # Keep doing this indefinitely
    while True:
        # Add 1 to the count
        count += 1

        # The 'yield' statement identifies this as a generator function.
        # Yield is like return, but it tells the generator function to freeze
        # in place and remember where it was ready for the next time it's
        # called
        yield count

# We run a generator function a little differently than a conventional
# function.  Here, we create an 'instance' of the generator function, and then
# we can work with that instance.
my_generator = keep_count_generator()

# Let's print the output of the generator function 5 times.  The 'next'
# statement is used to move to the next element of the iterator.  Here, that
# means the generator unfreezes and carries on until it hits another yield
# statement.
# I've not put these print statements in a for loop to make it clear what's
# happening.
print (next(my_generator))
print (next(my_generator))
print (next(my_generator))
print (next(my_generator))
print (next(my_generator))

# The above wouldn't work with a conventional function - every time the
# function is called, it would reset the count to 0, add 1 and then return
# a value of 1
def conventional_keep_count():
    count = 0

    while True:
        count += 1
        return count

a = conventional_keep_count()
print(a)
a = conventional_keep_count()
print(a)
a = conventional_keep_count()
print(a)
a = conventional_keep_count()
print(a)
a = conventional_keep_count()
print(a)
```

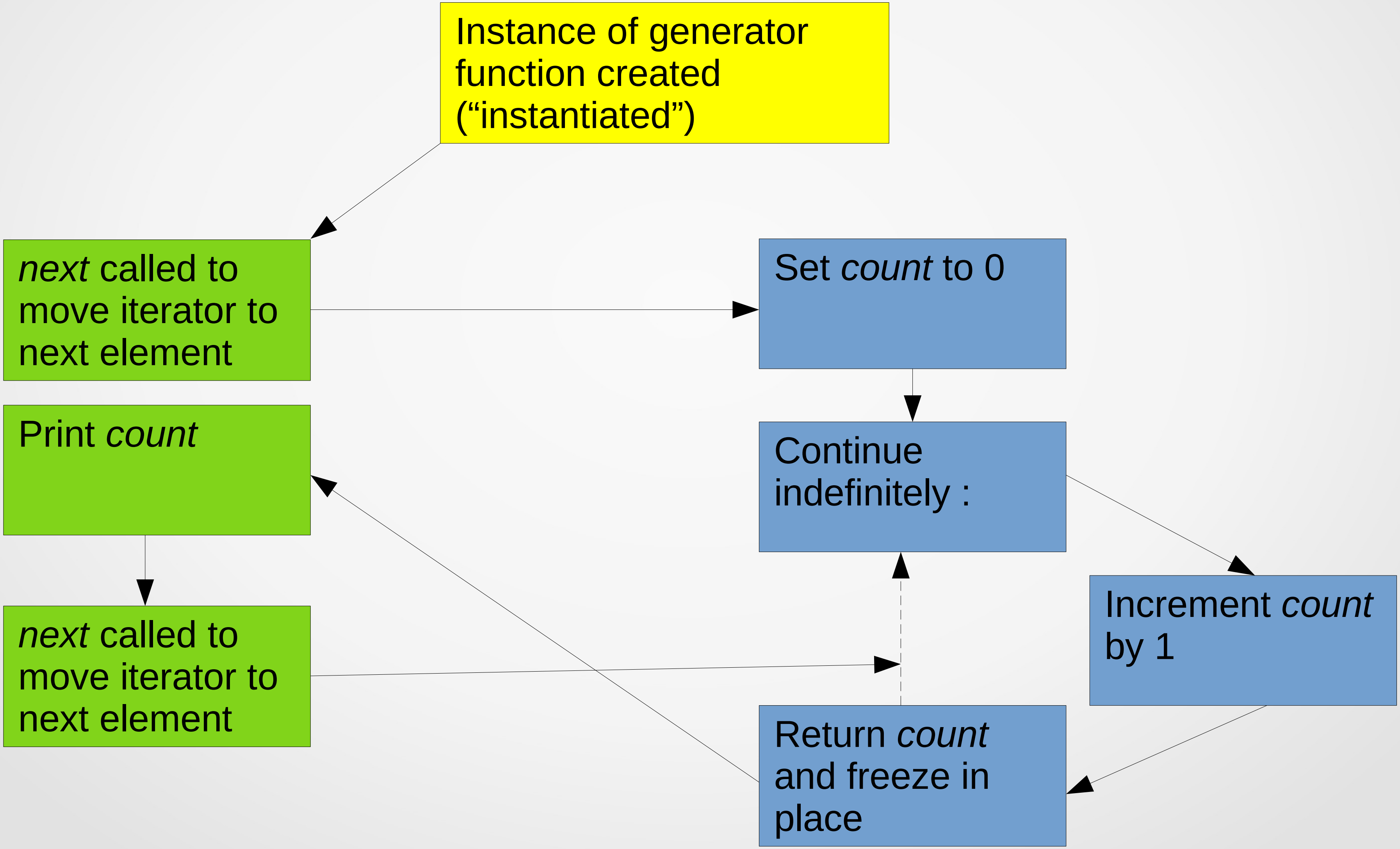


```
1
2
3
4
5
```



```
1
1
1
1
1
```

Generator Functions



Instance of generator function created (“instantiated”)

next called to move iterator to next element

Set *count* to 0

Continue indefinitely :

Increment *count* by 1

Return *count* and freeze in place

Print *count*

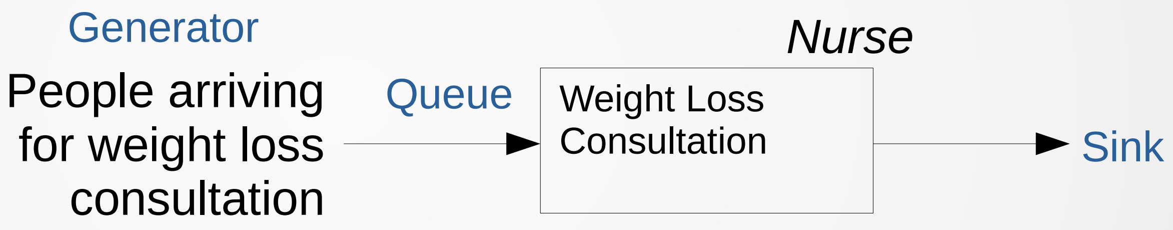
next called to move iterator to next element

Back to our simple example

Inter-Arrival Times :
- Arrivals waiting for weight loss consultation

Entities :
- Patients

Activity Times :
- Time spent in weight loss consultation



Generator

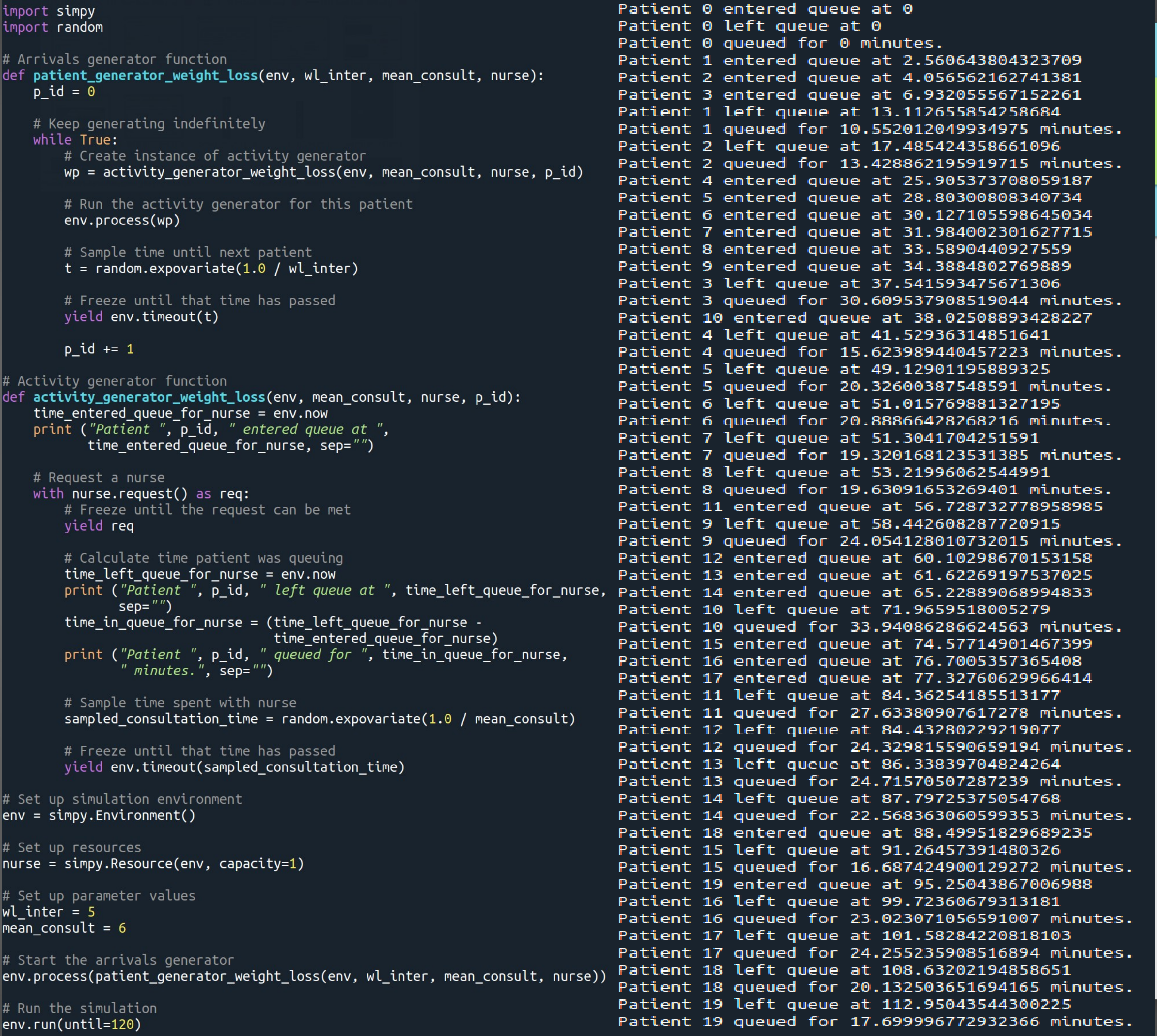
People arriving for weight loss consultation

Queue

Nurse

Weight Loss Consultation

Sink



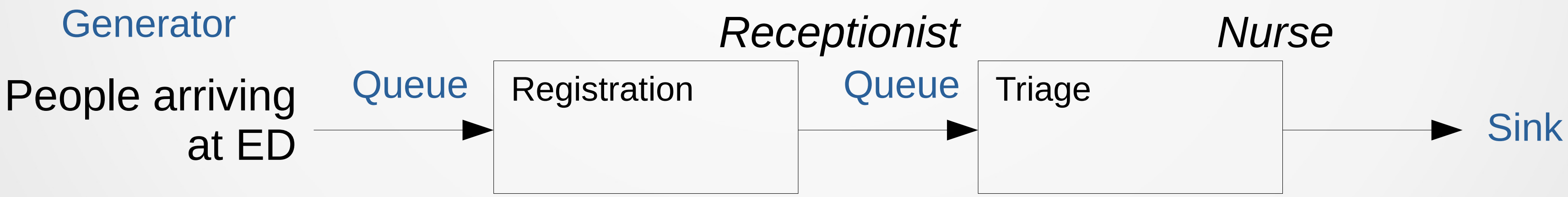
```python
import simpy
import random

# Arrivals generator function
def patient_generator_weight_loss(env, wl_inter, mean_consult, nurse):
    p_id = 0

    # Keep generating indefinitely
    while True:
        # Create instance of activity generator
        wp = activity_generator_weight_loss(env, mean_consult, nurse, p_id)

        # Run the activity generator for this patient
        env.process(wp)

        # Sample time until next patient
        t = random.expovariate(1.0 / wl_inter)

        # Freeze until that time has passed
        yield env.timeout(t)

        p_id += 1

# Activity generator function
def activity_generator_weight_loss(env, mean_consult, nurse, p_id):
    time_entered_queue_for_nurse = env.now
    print ("Patient ", p_id, " entered queue at ",
           time_entered_queue_for_nurse, sep="")

    # Request a nurse
    with nurse.request() as req:
        # Freeze until the request can be met
        yield req

        # Calculate time patient was queuing
        time_left_queue_for_nurse = env.now
        print ("Patient ", p_id, " left queue at ", time_left_queue_for_nurse,
               sep="")
        time_in_queue_for_nurse = (time_left_queue_for_nurse -
                                   time_entered_queue_for_nurse)
        print ("Patient ", p_id, " queued for ", time_in_queue_for_nurse,
               " minutes.", sep="")

        # Sample time spent with nurse
        sampled_consultation_time = random.expovariate(1.0 / mean_consult)

        # Freeze until that time has passed
        yield env.timeout(sampled_consultation_time)

# Set up simulation environment
env = simpy.Environment()

# Set up resources
nurse = simpy.Resource(env, capacity=1)

# Set up parameter values
wl_inter = 5
mean_consult = 6

# Start the arrivals generator
env.process(patient_generator_weight_loss(env, wl_inter, mean_consult, nurse))

# Run the simulation
env.run(until=120)
```

```
Patient 0 entered queue at 0
Patient 0 left queue at 0
Patient 0 queued for 0 minutes.
Patient 1 entered queue at 2.560643804323709
Patient 2 entered queue at 4.056562162741381
Patient 3 entered queue at 6.932055567152261
Patient 1 left queue at 13.112655854258684
Patient 1 queued for 10.552012049934975 minutes.
Patient 2 left queue at 17.485424358661096
Patient 2 queued for 13.428862195919715 minutes.
Patient 4 entered queue at 25.905373708059187
Patient 5 entered queue at 28.80300808340734
Patient 6 entered queue at 30.127105598645034
Patient 7 entered queue at 31.984002301627715
Patient 8 entered queue at 33.5890440927559
Patient 9 entered queue at 34.3884802769889
Patient 3 left queue at 37.541593475671306
Patient 3 queued for 30.609537908519044 minutes.
Patient 10 entered queue at 38.02508893428227
Patient 4 left queue at 41.52936314851641
Patient 4 queued for 15.623989440457223 minutes.
Patient 5 left queue at 49.12901195889325
Patient 5 queued for 20.32600387548591 minutes.
Patient 6 left queue at 51.015769881327195
Patient 6 queued for 20.88866428268216 minutes.
Patient 7 left queue at 51.3041704251591
Patient 7 queued for 19.320168123531385 minutes.
Patient 8 left queue at 53.21996062544991
Patient 8 queued for 19.63091653269401 minutes.
Patient 11 entered queue at 56.728732778958985
Patient 9 left queue at 58.442608287720915
Patient 9 queued for 24.054128010732015 minutes.
Patient 12 entered queue at 60.10298670153158
Patient 13 entered queue at 61.62269197537025
Patient 14 entered queue at 65.22889068994833
Patient 10 left queue at 71.9659518005279
Patient 10 queued for 33.94086286624563 minutes.
Patient 15 entered queue at 74.57714901467399
Patient 16 entered queue at 76.7005357365408
Patient 17 entered queue at 77.32760629966414
Patient 11 left queue at 84.36254185513177
Patient 11 queued for 27.63380907617278 minutes.
Patient 12 left queue at 84.43280229219077
Patient 12 queued for 24.329815590659194 minutes.
Patient 13 left queue at 86.33839704824264
Patient 13 queued for 24.71570507287239 minutes.
Patient 14 left queue at 87.79725375054768
Patient 14 queued for 22.568363060599353 minutes.
Patient 18 entered queue at 88.49951829689235
Patient 15 left queue at 91.26457391480326
Patient 15 queued for 16.687424900129272 minutes.
Patient 19 entered queue at 95.25043867006988
Patient 16 left queue at 99.72360679313181
Patient 16 queued for 23.023071056591007 minutes.
Patient 17 left queue at 101.58284220818103
Patient 17 queued for 24.255235908516894 minutes.
Patient 18 left queue at 108.63202194858651
Patient 18 queued for 20.132503651694165 minutes.
Patient 19 left queue at 112.95043544300225
Patient 19 queued for 17.699996772932366 minutes.
```

More than one sequential activity - Linear

In most DES models, we'd have our more than one activity in our system.

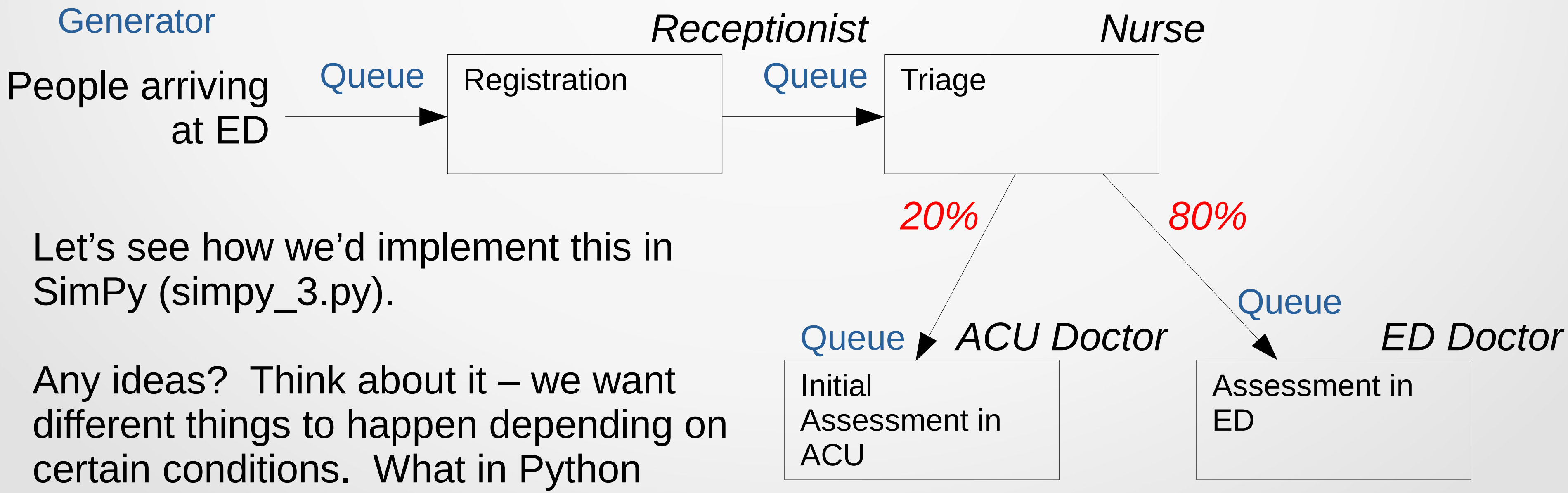
Let's first consider a simple *linear* example, where patients queue for one activity, and then queue for a second activity.



In the above example, we have two activities that patients queue for sequentially, and two types of resource – receptionists and nurses. Let's see how we'd implement that in SimPy (simpy_2.py).

More than one sequential activity - Non-Linear

Often in real world systems, not everything is linear. Different things might happen depending on certain conditions. For example, after triage a patient might be referred to a Ambulatory Care to diagnose and treat the patient without being admitted to the hospital overnight.



Let’s see how we’d implement this in SimPy (simpy_3.py).

Any ideas? Think about it – we want different things to happen depending on certain conditions. What in Python could we use for that?

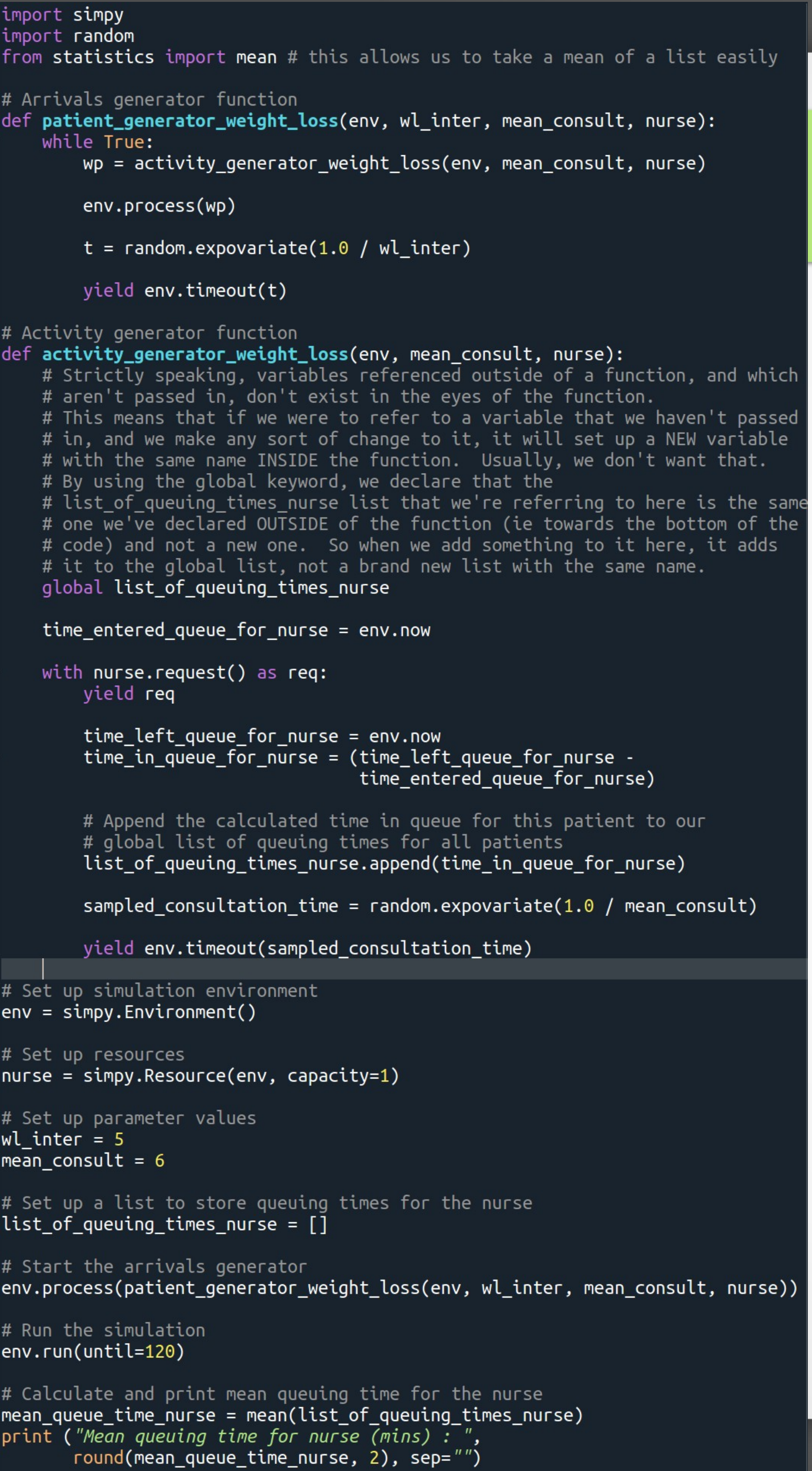
Storing Results

Let’s say we want to store results over the simulation run so we can, for example, calculate the average queuing time across patients.

One way we could do this would be to set up a list to store the results we’re interested in, and then add each patient’s results to the list.

If we do that, we need to make use of the *global* keyword.

Let’s look at an example (simpy_4.py).

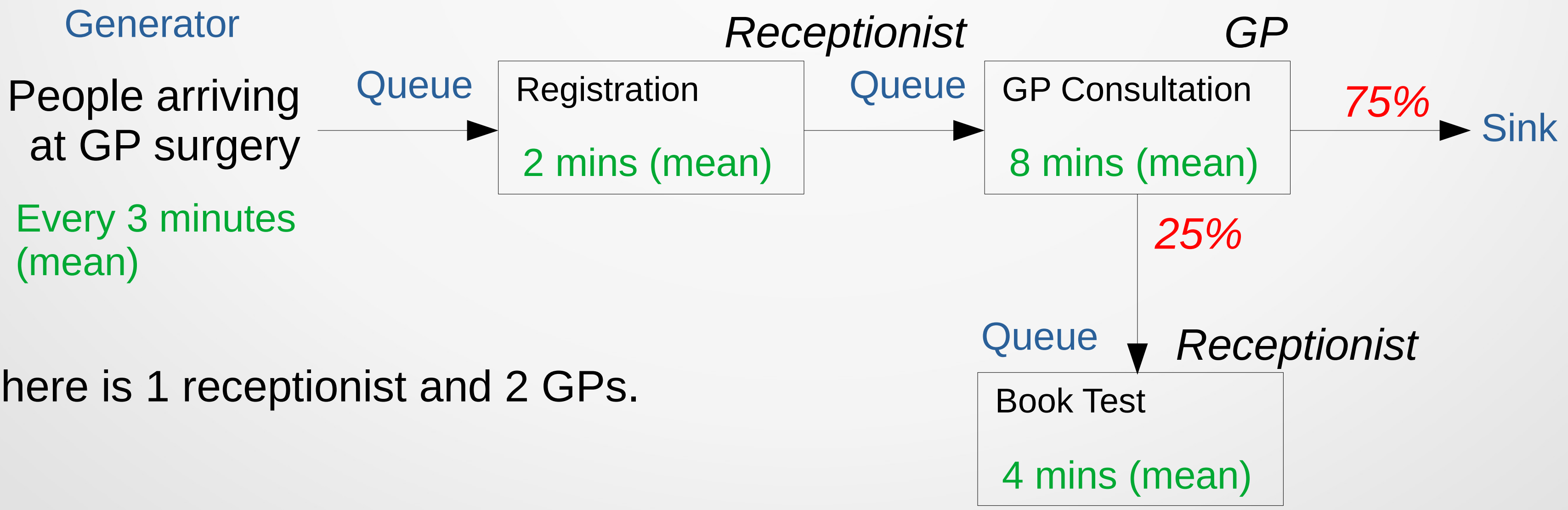


```python
import simpy
import random
from statistics import mean # this allows us to take a mean of a list easily

# Arrivals generator function
def patient_generator_weight_loss(env, wl_inter, mean_consult, nurse):
    while True:
        wp = activity_generator_weight_loss(env, mean_consult, nurse)

        env.process(wp)

        t = random.expovariate(1.0 / wl_inter)

        yield env.timeout(t)

# Activity generator function
def activity_generator_weight_loss(env, mean_consult, nurse):
    # Strictly speaking, variables referenced outside of a function, and which
    # aren't passed in, don't exist in the eyes of the function.
    # This means that if we were to refer to a variable that we haven't passed
    # in, and we make any sort of change to it, it will set up a NEW variable
    # with the same name INSIDE the function.  Usually, we don't want that.
    # By using the global keyword, we declare that the
    # list_of_queuing_times_nurse list that we're referring to here is the same
    # one we've declared OUTSIDE of the function (ie towards the bottom of the
    # code) and not a new one.  So when we add something to it here, it adds
    # it to the global list, not a brand new list with the same name.
    global list_of_queuing_times_nurse

    time_entered_queue_for_nurse = env.now

    with nurse.request() as req:
        yield req

        time_left_queue_for_nurse = env.now
        time_in_queue_for_nurse = (time_left_queue_for_nurse -
                                   time_entered_queue_for_nurse)

        # Append the calculated time in queue for this patient to our
        # global list of queuing times for all patients
        list_of_queuing_times_nurse.append(time_in_queue_for_nurse)

        sampled_consultation_time = random.expovariate(1.0 / mean_consult)

        yield env.timeout(sampled_consultation_time)

# Set up simulation environment
env = simpy.Environment()

# Set up resources
nurse = simpy.Resource(env, capacity=1)

# Set up parameter values
wl_inter = 5
mean_consult = 6

# Set up a list to store queuing times for the nurse
list_of_queuing_times_nurse = []

# Start the arrivals generator
env.process(patient_generator_weight_loss(env, wl_inter, mean_consult, nurse))

# Run the simulation
env.run(until=120)

# Calculate and print mean queuing time for the nurse
mean_queue_time_nurse = mean(list_of_queuing_times_nurse)
print ("Mean queuing time for nurse (mins) : ",
       round(mean_queue_time_nurse, 2), sep="")
```

Exercise 1

Build a DES model in SimPy of the below system. Run for 8 hours of simulated time. Add functionality to the model to calculate and print the mean queuing times for each of the three queues, along with the mean time in system (from point of arrival to the surgery, to point of departure). For extra points, additionally plot a bar chart showing these times.

You have 1 hour. You may choose to work in small groups if you prefer.



Generator

People arriving at GP surgery

Every 3 minutes (mean)

Queue

Receptionist

Registration

2 mins (mean)

Queue

GP

GP Consultation

8 mins (mean)

75%

Sink

25%

Queue

Receptionist

Book Test

4 mins (mean)

There is 1 receptionist and 2 GPs.

Multiple Generators

It’s quite possible we’ll have more than one way for our entities to enter our system in our model. They may or may not go through the same processes.

For example, patients might arrive at an ED by self-presenting or by ambulance, but the same things might happen to them regardless.

Or, in our GP surgery example from the first session, people may be coming in to the same place and using the same resources but for different reasons (e.g. coming to see nurse for test, or for weight loss clinic).

Multiple Generators in SimPy

Either way, it’s easy to capture multiple entry points into our system in SimPy – we just set up more than one entity generator!

Let’s look at an example (simpy_5.py).

Multiple Runs

In a stochastic model, it is important that we do not just run a model once if we’re looking to draw insights from our results.

This is because every run of the simulation will have different random samples for inter-arrival times, activity times etc.

What if you had a run with unusually long activity times sampled (a run of “bad luck”)? Or unusually long inter-arrival times (a run of “good luck”)?

We need to run a stochastic simulation many times and take summary statistics over the results from each run to get more representative results from the model.

Multiple Runs in SimPy

In SimPy, to run a simulation multiple times, we put *all* of the **setup** (including the establishment of the environment) in a **for loop**, with the range specified as the number of runs we want.

This means that the simulation resets and starts anew after every run, so if we want to store individual simulation run results (which we need to calculate averages) we either need to :

- have a global list outside of the loop that stores the results from each run

OR

- write the results of each run to file

Let's look at an example (simpy_6.py), where we've added multiple runs to our previous simulation model.

Warm Up

A potential problem in our DES models is that, by default, we assume that the system is empty of entities (e.g. patients) when the system starts.

In some cases, that might be fine – for example, if we're simulating a day of Minor Injury Unit activity, where patients turn up from when it opens and there's nobody left when it closes.

But imagine we're modelling an ED or a hospital more generally. These systems are never empty, and so if we run our model starting from empty, then our results are going to be skewed by the fact that early on in the simulation, there were no or very few people in the model.

Fortunately, there's a simple solution – we can use a *warm up period*.

Warm Up Period

During a warm up period, our model runs, but it doesn’t collect any results. The warm-up period allows the simulation to get to a more representative state before we start collecting results.

The length of our warm up period will vary depending on the system we’re modelling. We might choose to just set a sufficiently lengthy warm up period to ensure the system is more representative once results start to be taken, or we might run the simulation multiple times and track the results over time to see when a steadier state appears to be reached.

Either way, we can easily add a warm up period to our SimPy models by simply using conditional logic to ensure that results are not collected until the simulation time has passed the warm up period. Let’s look at an example (simpy_7.py).

Containers

So far, we’ve considered resources as being discrete identifiable entities. But what if our resource was continuous, or if we wanted to model a homegenous mass of resource.

For example, imagine we were modelling the fuelling of a car, and our resource was petrol.

Or, maybe rather than model number of nurses, we want to model simultaneous nurse time in use, from which we take from and add back to a pool of time as incoming activity fluctuates.

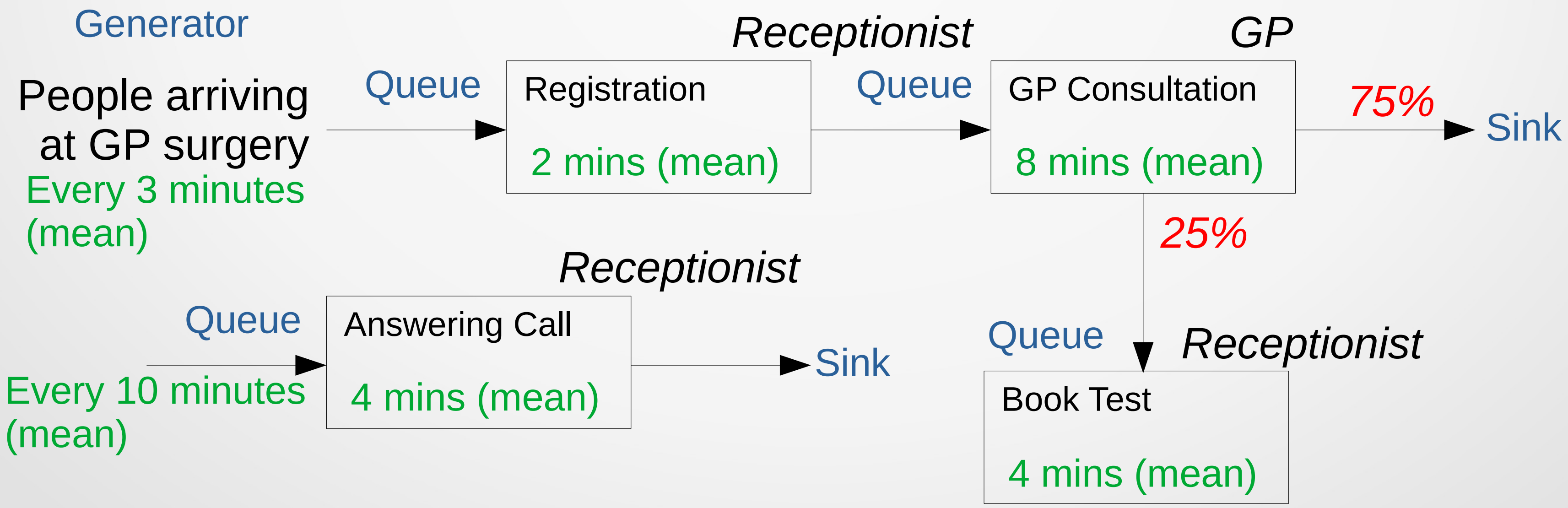
In SimPy, rather than use the Resource class, we can use the *Container* class for these kinds of problems.

Let’s look at an example (simpy_8.py).

Exercise 2

Extend the DES model you built in Exercise 1 to include :
- calls coming in to reception on average every 10 minutes, and which last an average of 4 minutes, answered by the receptionist. Don’t worry about adding any results for call queues, or adding these calls to times in system results. We just want to model the drain on the receptionist resource for other activities (registering patients and booking tests)
- a warm up period of 3 hours before the 8 hours results collection period
- functionality to run the simulation multiple times – have the simulation run 100 times, with average results taken over these runs

You have 40 minutes. We’ll share the solution at the end of this session.



There is 1 receptionist and 2 GPs.