SimPy Part 2

SimPy for Discrete Event Simulation Part 2

Dr Daniel Chalk

Object Oriented SimPy

Up to this point, we've shown you a largely non-object oriented way of putting together SimPy models.

One thing you've probably noticed (and got increasingly fed up with) is passing around parameter values, resources and simulation environments between generator functions. Which means that when you add more detail to the model, things quickly become fiddly.

But there's a better way. And as modern coders, you should be looking to make your code object-oriented wherever possible.

Let's consider how we'd do this with SimPy.

Object Oriented SimPy

CLASS : g

Attributes
global_parameter_A = 5
global_parameter_B = 2
global_parameter_C = 72

Methods

We have a class that stores global variable values that we'll need across the model (things like mean inter-arrival times, mean process times, simulation duration, number of runs etc). We don't create an instance of this class – we just refer to the Class blueprint itself.

CLASS : Entity_1

Attributes
entity_attribute_A
entity_attribute_B
entity_attribute_C

Methods
__init__

We have a class for each of the entity types flowing through our model. Each entity may have attributes (e.g. patient ID) and may have specific methods, or may just have a constructor.
Every time a new entity (eg patient) arrives, we create a new instance of this class.

CLASS : Model

Attributes
env
resource_1
resource_2

Methods
__init__
entity_generator
set_of_processes
run

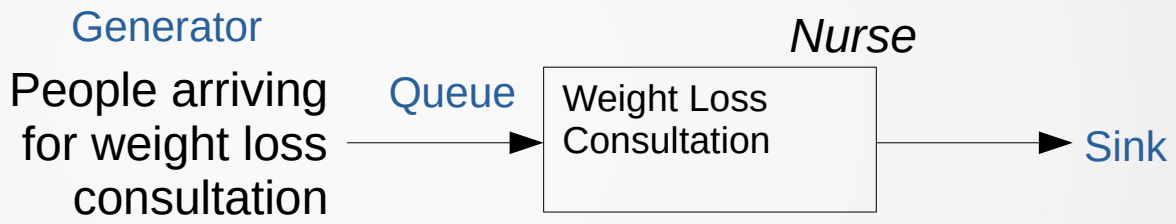
We have a class to represent our model. The attributes of this class will include the simulation environment and any resources. Methods will include the constructor, one or more entity generators, one or more functions describing sequences of events that will happen to the entities, and a run function that will start the entity generators and run the model for the required amount of time.

Simple Example

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

Entities :
- Patients

Activity Times :
- Time spent in weight loss consultation



Simple Example

CLASS : g

Attributes
wl_inter = 5
mean_consult = 6
number_of_nurses = 1
sim_duration = 120
number_of_runs = 10

Methods

CLASS : Weight_Loss_Patient

Attributes
p_id : integer

Methods
__init__()

CLASS : GP_Surgery_Model

Attributes
env : simpy.Environment()
nurse : simpy.Resource()

Methods
__init__()
generate_wl_arrivals()
attend_wl_clinic(patient)
run()

Let’s see how we’d translate this into code (simpy_oo_1.py)

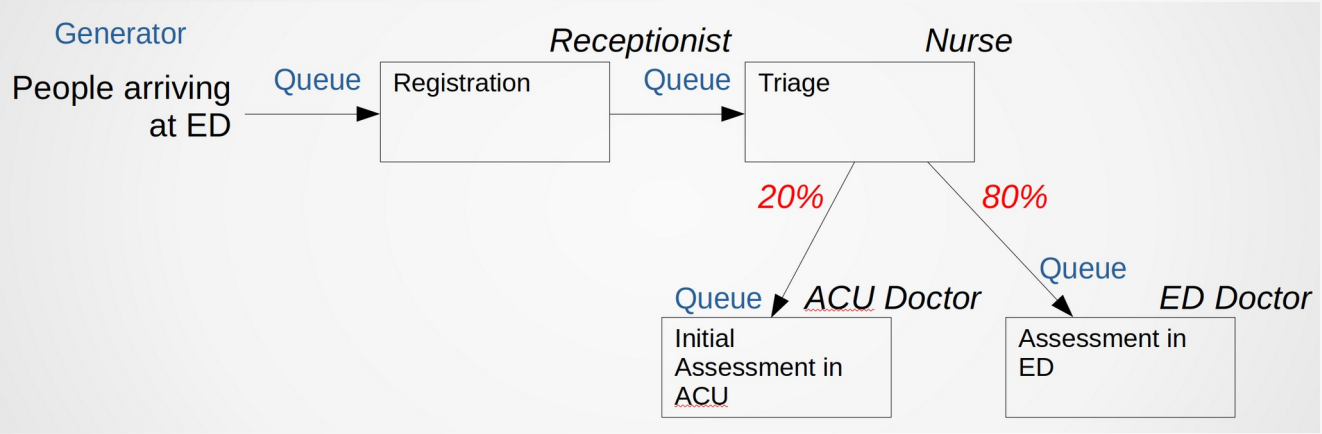
Capturing Trial Results

To store trial results in our new object-oriented framework, we could do the following :
- add a run number attribute to the GP_Surgery_Model class, and pass in a run number when we instantiate this class on each run, allowing us to store run number against run results
- add a Pandas DataFrame as an attribute to the GP_Surgery_Model class that stores the various results we want to capture in each run
- add methods to the GP_Surgery_Model class that calculate the results we want to store
- add a method to the GP_Surgery_Model class that writes the results from the run to file
- call these additional methods from the run() method after each simulation run
- create a new class that holds attributes and methods for calculating and printing results over the trial (batch of runs)
- create a file with appropriate column headers before we start running a batch of runs
- instantiate the new class for calculating trial results after the batch of runs has completed, and call the relevant functions.

Let’s look at how we would do this for our simple model, if we wanted to record average queuing time for the nurse in each run, and then take an average over runs in the trial (simpy_oo_2.py).

Exercise 1

Your task is to write an **Object Oriented** version of the below model in SimPy. You may recognise this from simpy_3.py from the last session.



Mean inter-arrival : 8 minutes
Mean registration : 2 minutes
Mean triage : 5 minutes
Mean ED Assessment : 30 mins
Mean ACU Assessment : 60 mins
Probability of going to ACU : 20%
No. receptionists : 1
No. nurses : 2
No. ED Doctors : 2
No. ACU Doctors : 1
Run for 48 hours after a 24 hour warm up period.
Run 100 times

You need to ensure that the model stores the queuing times for each queue for each patient (you don't need to store start and end queue times), and records average queuing times for each run in a file, and calculates and prints average queuing time results over the trial. It is recommended that you store the queuing time results for a patient in a Pandas DataFrame stored in the model class at the end of the patient's journey through the ED, alongside their patient ID. However, as you have a branching path here, some patients won't have queuing times for the ACU assessment, whilst the others won't have queuing times for the ED assessment. I recommend that you replace the missing queuing time in each instance with a "NaN" (Not a Number). You can generate a NaN by casting the string "nan" as a float. NaNs are ignored by Pandas when calculating things like the mean of a column.

I also want you to deal with the branching path by having attributes in the patient class that store the probability of a patient going to the ACU and a flag to indicate whether they are going to the ACU, and a method in the patient class that determines whether this patient will go to the ACU, and which is called after the patient is created, and before they start their journey through the ED (so, in essence, whether they go to the ACU is pre-destined at the point of their arrival).

Make sure you include a warm up period as specified above, so that results aren't stored within the warm-up period (patient journeys that start within the warm up period but end after the warm up period can be counted). Don't forget to ensure that your simulation runs for the warm up period in addition to the simulation run period.

You have 2 hours to write this model, and you should work in small groups.