NIHR | Applied Research Collaboration South West Peninsula

@penCHORD_UoE
@peninsula_ARC

**Module 3 : Modelling Pathway and Queuing Problems**
Session 3C : SimPy for Discrete Event Simulation Part 2
Dr Daniel Chalk

"SimPy the Best"

HSMA 5

#hsma5isalive

# 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**

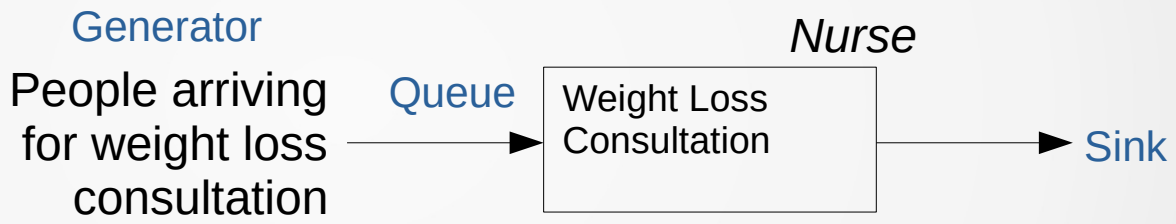
We have a class to represent our model. The attributes of this class will include the simulation environment and any resources. Methods will include the constructor, one or more entity generators, one or more functions describing sequences of events that will happen to the entities, and a run function 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 :**
- WL Clinic 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)

Note : it may appear that this code is much lengthier and more complicated than the original non-OO code for this model. And it is. But this is a very simple example model – for more complex real-world models (or models that we add to over time), you'll see the benefits of this approach quite quickly.

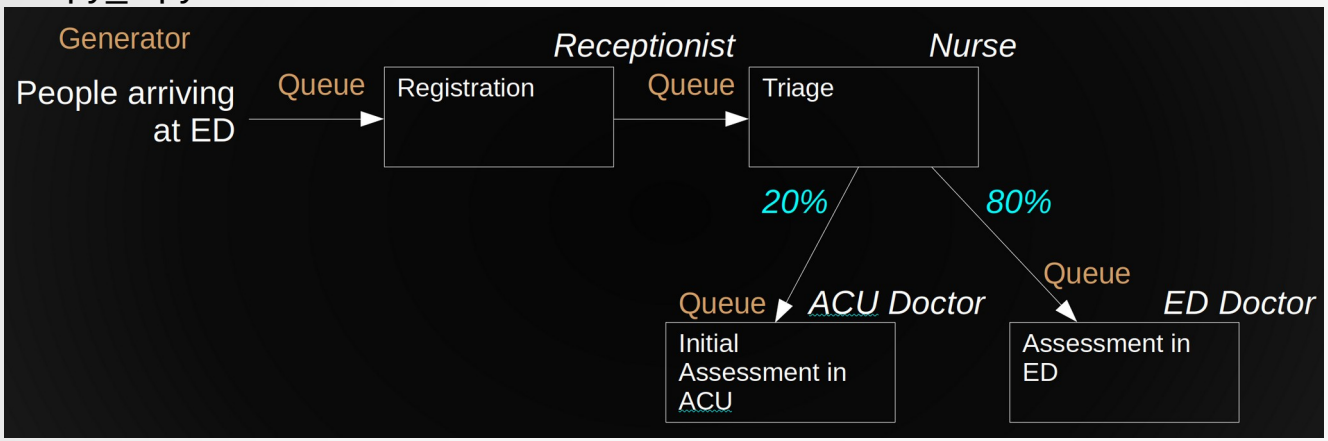
# Capturing Trial Results

To store trial results in our new object-oriented framework, we could do the following :
1. 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
2. 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
3. add methods to the GP_Surgery_Model class that calculate the results we want to store
4. add a method to the GP_Surgery_Model class that writes the results from the run to file
5. call these additional methods from the run() method after each simulation run
6. create a new class that holds attributes and methods for calculating and printing results over the trial (batch of runs)
7. create a file with appropriate column headers before we start running a batch of runs
8. 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 1.5 hrs. Work in your groups. Make sure you take a comfort break in there too.

# Priority-Based Queuing

So far, we've assumed our queues follow a FIFO (First in First Out) Policy.  This means that our entities are drawn from the queue in the order in which they arrive.

But in healthcare systems, very often there is an element of prioritisation in a real world queue.  Typically this represents the severity of the patient's condition.

We can easily build in priority-based queuing in our SimPy models using something known as a PriorityResource.

# PriorityResource

Up until now, for resources in our models, we've only used the standard *Resource* class in SimPy.

But if we want resources to see entities in some form of priority order for a queue, we need to use SimPy's *PriorityResource* class instead.

We instantiate a PriorityResource in exactly the same way as a standard Resource – by specifying the environment in which it will live, and the capacity)

```
# If we want a queue where higher priority entities are seen first,
# then the resource they queue for needs to be a PriorityResource
self.ed_doctor = simpy.PriorityResource(
    self.env, capacity=g.number_of_ed_doctors)
self.acu_doctor = simpy.PriorityResource(
    self.env, capacity=g.number_of_acu_doctors)
```

# How it Works

A PriorityResource will look at the entities queuing for it and will look at a specific integer variable (that we specify) associated with that entity to determine the “priority” of that entity in the queue – **lower** values indicating a **higher** priority.

```
# The PriorityResource will see entities (in this case patients)
# according to their priority - an integer value, where the lower the
# integer, the higher the priority
self.priority = 1
```

```
# Method to determine the patient's priority.  Here we just randomly
# select a priority value, but obviously this could include any logic you
# like
def determine_priority(self):
    self.priority = random.randint(1,5)
```

```
# Now the patient has been triaged, we can assign their priority
# to determine how quickly they'll be seen either by the ED doctor
# or the ACU doctor
patient.determine_priority()
```

# How it Works

As well as setting up a resource as a PriorityResource, and setting up an integer variable in the entity class that will store that entity's priority value, we also need to tell the PriorityResource which variable to look at when it's drawing entities from the queue.

We do this at the point at which we request the resource :

```
# Request an ACU doctor - now that ACU doctor is a
# PriorityResource, we also specify the value to be used to
# determine priority.  Here, that's the priority attribute of the
# patient object.
with self.acu_doctor.request(priority=patient.priority) as req:
```

In this example, a patient with a priority of 3 would be seen before a patient with priority 4, even if the patient with priority 4 joined the queue first.

# Example

Let’s have a look at an example of priority queuing in action (simpy_oo_priority_resource.py).

# Resource unavailability

So far in our models, we've assumed that, outside of working on our modelled activities, our modelled resources are always available.

But that won't always be the case in the real world. Resources may not always be "on shift", or may be called off to other areas of the system (e.g. different parts of the hospital).

We can model this in SimPy by "obstructing" a resource for a certain amount of time.

# Resource unavailability

Let's consider an example using our ED model. Let's imagine we want to model that an ED Doctor isn't always available for our part of the system. This may not be *the same* ED doctor in each case – it might just represent that we're down an ED doctor for an amount of time.

We could set up an element of randomness around this (particularly if you're modelling a resource being "called away"), but in this example we'll assume that we're emulating shift patterns, so we'll use fixed periods of unavailability. Specifically, we'll set things up so that every 8 hours, an ED doctor is unavailable for 4 hours. We'll create some new attribute values in class *g*.

```
unavail_time_ed_doctor = 240
unavail_freq_ed_doctor = 480
```

# Resource unavailability

Next, we'll set up a new entity generator.  This one won't represent patients coming into the system, but instead will represent an "abstract" entity that takes priority over everything else and will "obstruct" the doctor resource from working in our system.

```
# A method to obstruct the ed doctor(s) to emulate other non-modelled
# tasks or other unavailability
def obstruct_ed_doctor(self):
    while True:
        print ("An ED Doctor will be removed around time",
               f"{self.env.now + g.unavail_freq_ed_doctor:.1f}")
```

# Resource unavailability

The first thing we'll get this new entity generator to do is freeze for the time we've specified as our frequency (8 hours here). In other words, don't do anything for 8 hours. This represents the ED doctor being **on shift**.

```
# Freeze the function for the unavailability frequency period.
# This could represent time on shift (the time the doctor is
# available).  Here, we use a fixed value to reflect a set shift,
# but you could also sample from a probability distribution.
yield self.env.timeout(g.unavail_freq_ed_doctor)
```

# Resource unavailability

Next (once this time has elapsed), we're going to request an ED doctor (which we already set up as a PriorityResource) with a priority of -1. This means that this will take top priority, because all of our patients have priorities between 1 and 5, so this will always be lower (and therefore higher priority) than any of them. We'll then hold the ED doctor for the amount of time we set up as being our period of "unavailability".

Important – this doesn't mean that when this generator requests an ED doctor that it'll get one immediately. It'll just get one as soon as one is available (ie finished with a patient). This is likely what happens in your real world system – doctors don't tend to walk out mid-activity because their shift is over.

```
# Once this time has elapsed, request an ed doctor with a priority
# of -1 (so that we know this will get the top priority, as none
# of our patients will have a negative priority), and hold them
# for the specified unavailability amount of time
with self.ed_doctor.request(priority=-1) as req:
    # Freeze the function until the request can be met (this
    # ensures that the doctor will finish what they're doing first)
    yield req

    print ("An ED Doctor is unavailable. They'll be back at time",
           f"{self.env.now + g.unavail_time_ed_doctor:.1f}")

    yield self.env.timeout(g.unavail_time_ed_doctor)
```

# Resource unavailability

The only other thing we need to do is start up this new obstruct generator in the run method :

```
def run(self):
    # Start entity generators
    self.env.process(self.generate_ed_arrivals())
    self.env.process(self.obstruct_ed_doctor())
```

Let’s have a look at it in action (simpy_oo_unavailability.py)