Module 3 : Modelling Pathway and Queuing Problems

Session 3A : Introduction to Discrete Event Simulation

Dr Daniel Chalk

"It's all part of the process"

NIHR | Applied Research Collaboration South West Peninsula

@peninsula_ARC #HSMA4

HSMA 4

HSMA 4 Everyone

What is Discrete Event Simulation?

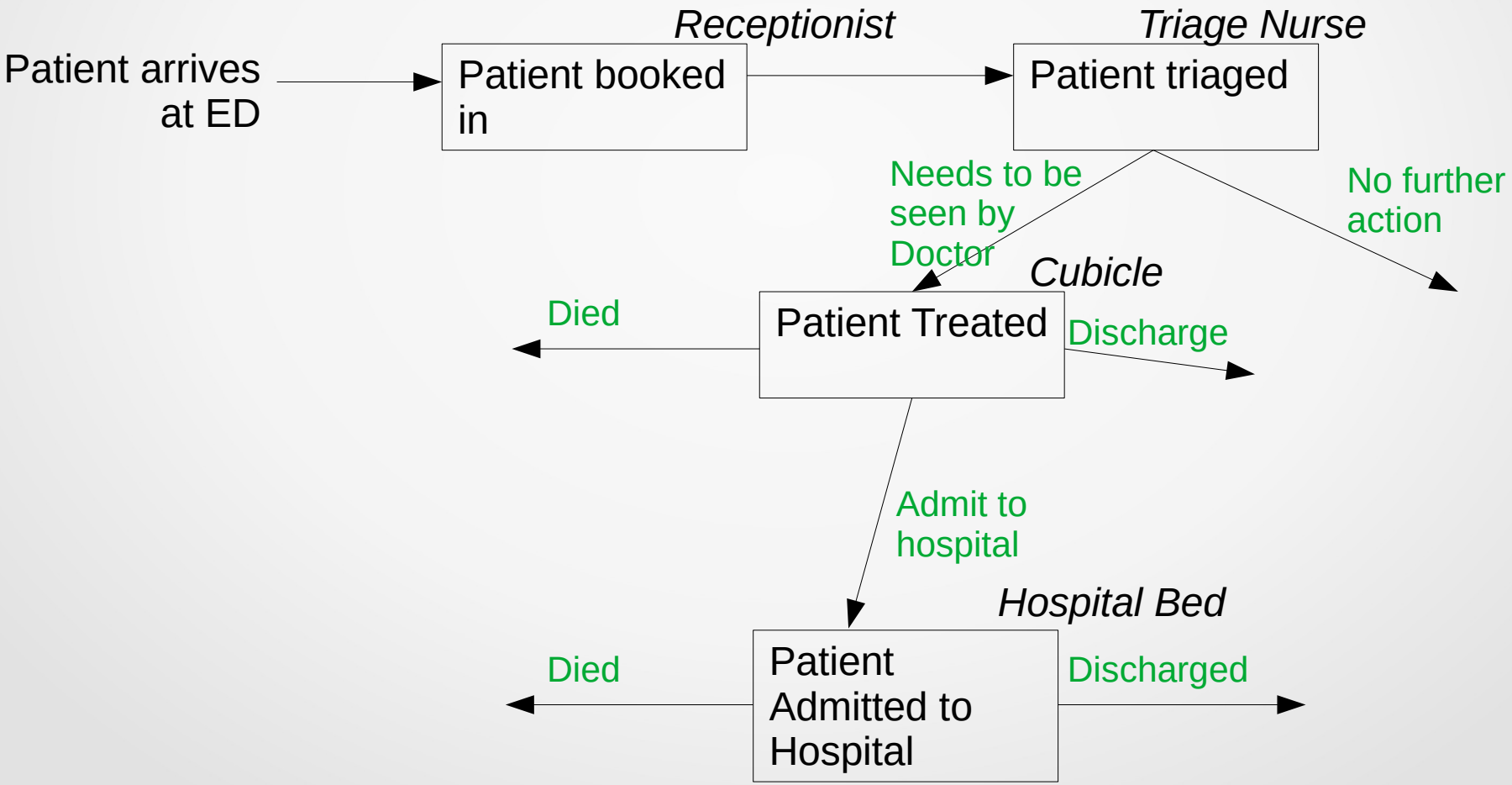
Discrete Event Simulation (DES) is a way of modelling *queuing problems*.

In a DES, *entities* flow through (and queue for) *discrete sequential processes* that use *resources*.

DES is typically used to model processes and pathways. For example, what happens to patients when they arrive at the Emergency Department.

Therefore, DES is useful for asking “what if?” questions about process / pathway changes.

An Example


Patient arrives at ED
Receptionist
Patient booked in
Triage Nurse
Patient triaged
Needs to be seen by Doctor
No further action
Cubicle
Patient Treated
Died
Discharge
Admit to hospital
Hospital Bed
Patient Admitted to Hospital
Died
Discharged

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)

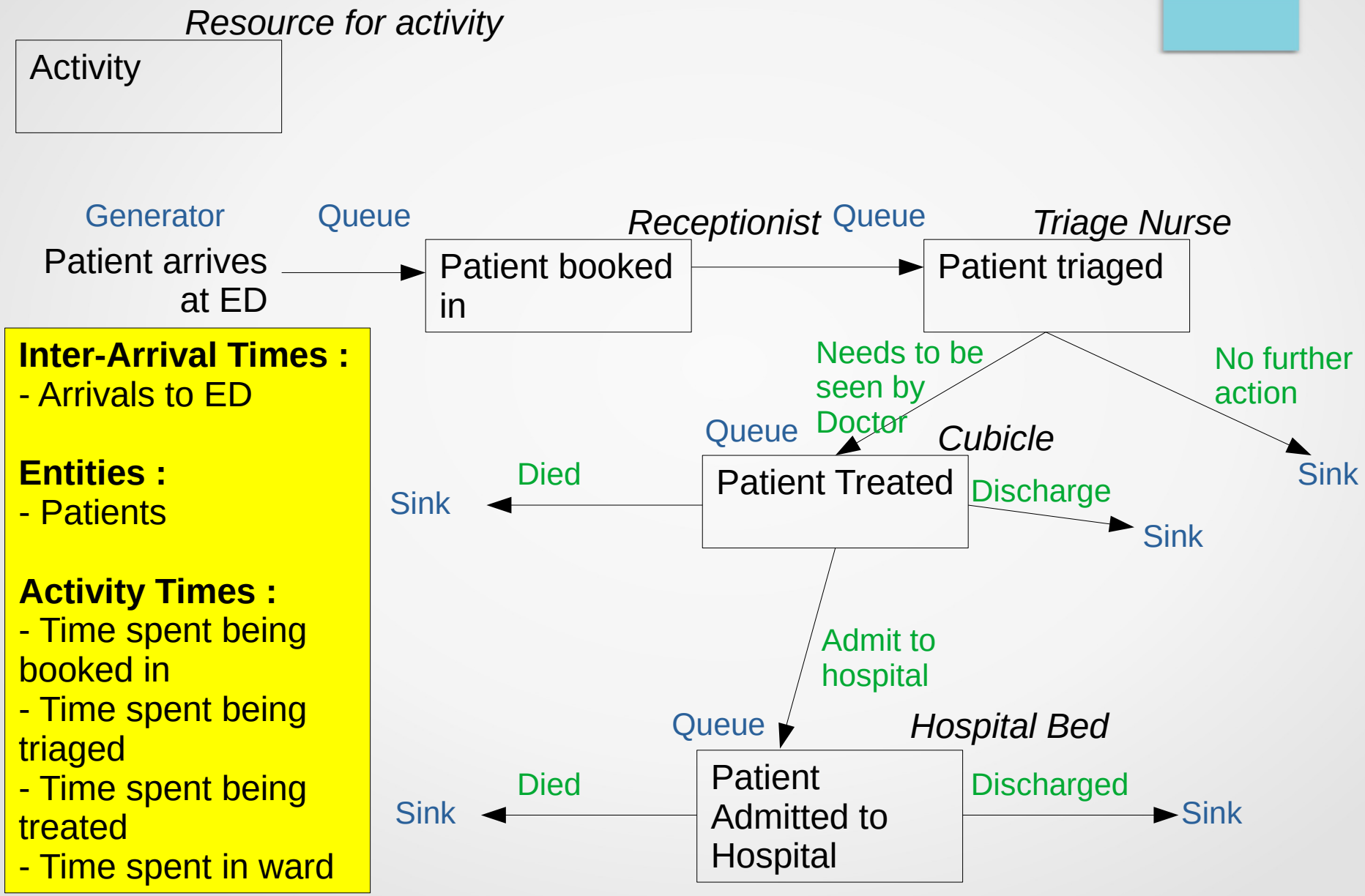
Activity / Server Time represents the amount of time it takes for an activity to happen to an entity.

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.

An Example



Inter-Arrival Times :
- Arrivals to ED

Entities :
- Patients

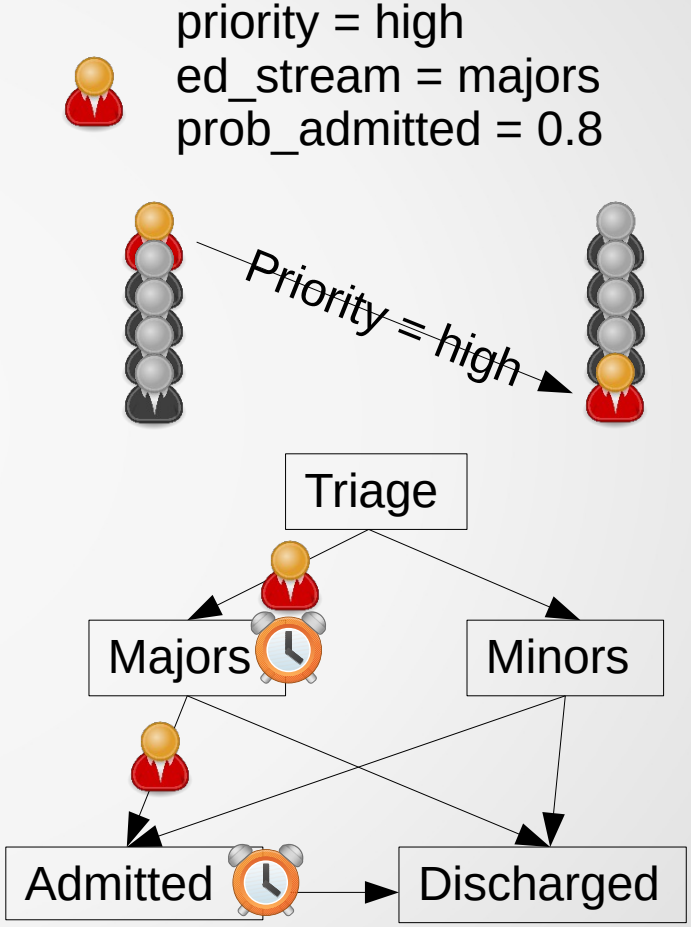
Activity Times :
- Time spent being booked in
- Time spent being triaged
- Time spent being treated
- Time spent in ward

Entities

Each entity may have certain “attributes” that will help determine its journey through the modelled system. For example :

- whether it goes down path A or B
- how long it spends in an activity
- its priority in a queue for an activity

There may be more than one type of entity in a model at the same time. For example, patients in a clinic, their test results, and phone calls into the clinic are all entities that we may want to capture when modelling the clinic.



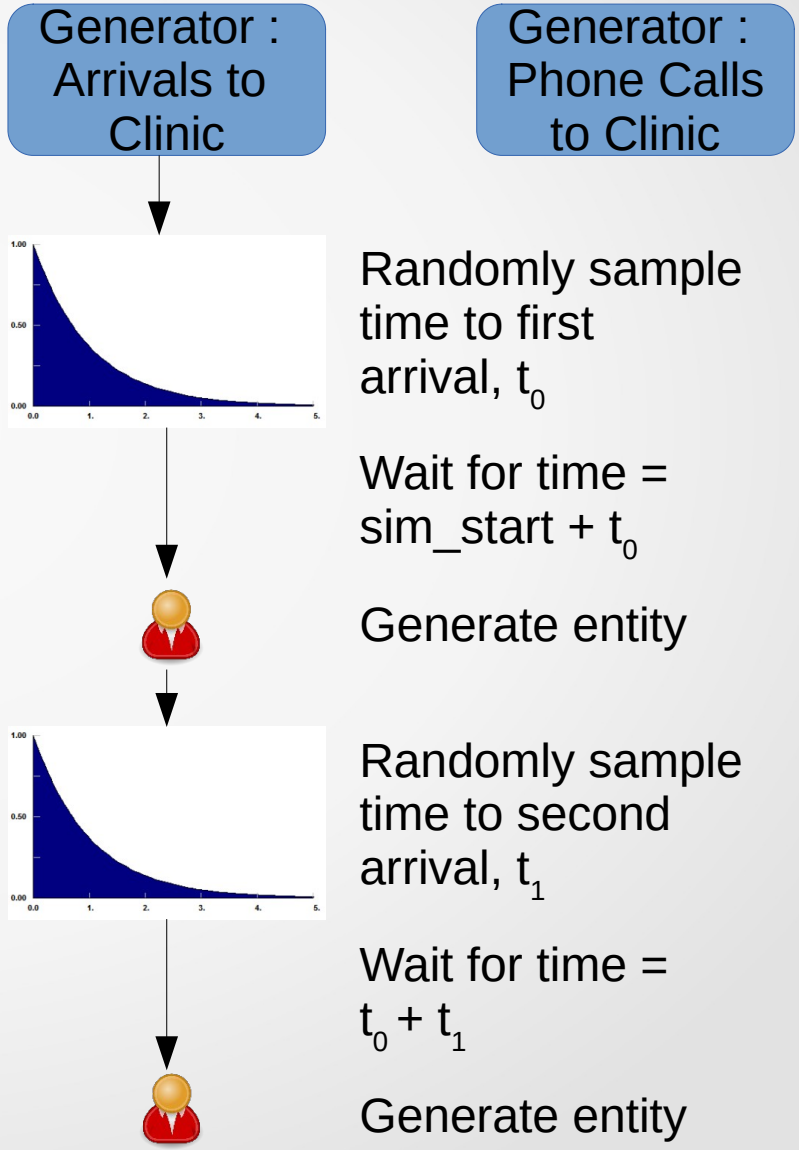
Generators and Inter-Arrival Times

A generator creates new entities to bring into the system. The rate at which new entities are generated is determined by an inter-arrival time.

The inter-arrival time determines the time between one entity being generated, and the next one being generated.

Inter-arrival times may be fixed, but are typically sampled (drawn) stochastically (randomly) from a distribution to capture variability (even if the variability is small)

An Exponential Distribution is often used to sample inter-arrival times. More than one distribution may be used for the same generator (e.g. for different times of the day, day of week etc). You may also have more than one generator in a system.



Queues

Each activity in a Discrete Event Simulation has an associated queue. The queue holds entities whilst they wait for the activity to become available for them.

Each queue has a *queuing policy*. This determines the order in which entities are released from the queue into the activity. The two most common queuing policies are:

- First In First Out (FIFO) : entities are seen in the order they arrive. This is the default.
- Priority-based : entities are seen according to some priority attribute. Ties often resolved using FIFO

First in, First Out (FIFO)

Priority-based

Reneging, Balking and Jockeying

Not all queues run “as planned”. We may wish to model behaviours where entities stop waiting, switch queues, or never join the queue in the first place.

Reneging refers to an entity removing themselves from a queue after a certain amount of time has elapsed.

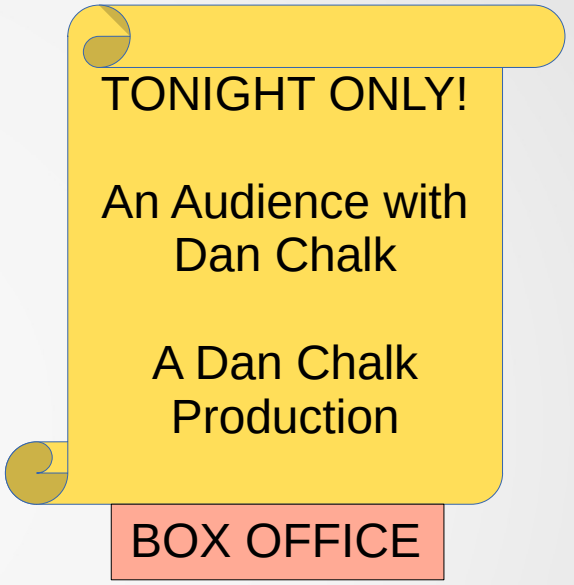
- e.g person not willing to wait any longer
- e.g test sample no longer being viable

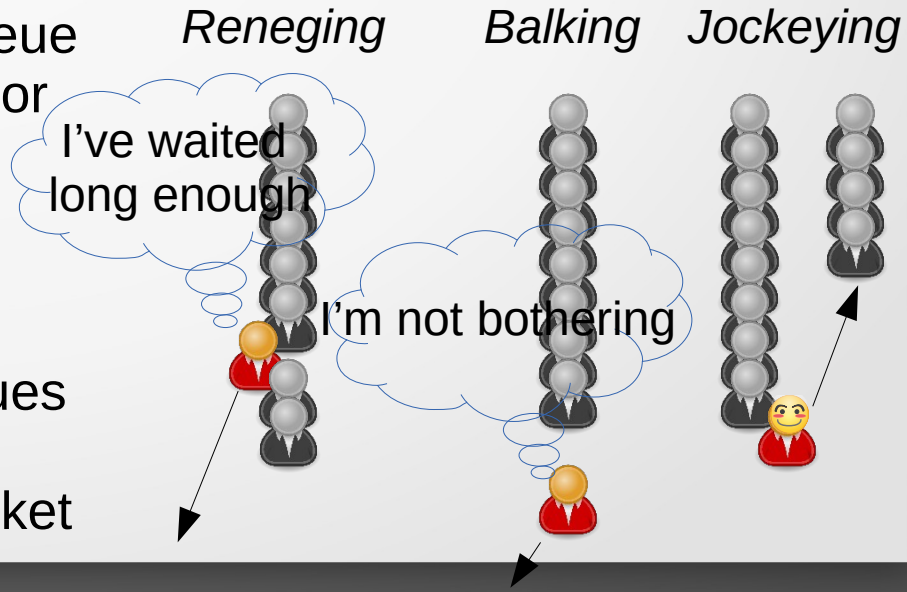
Balking refers to an entity not entering a queue in the first place because of the length and / or capacity of the queue.

- e.g person seeing long queue
- e.g no capacity in waiting room

Jockeying refers to an entity switching queues in the hope of reducing queuing time.

- e.g switching till queues at the supermarket





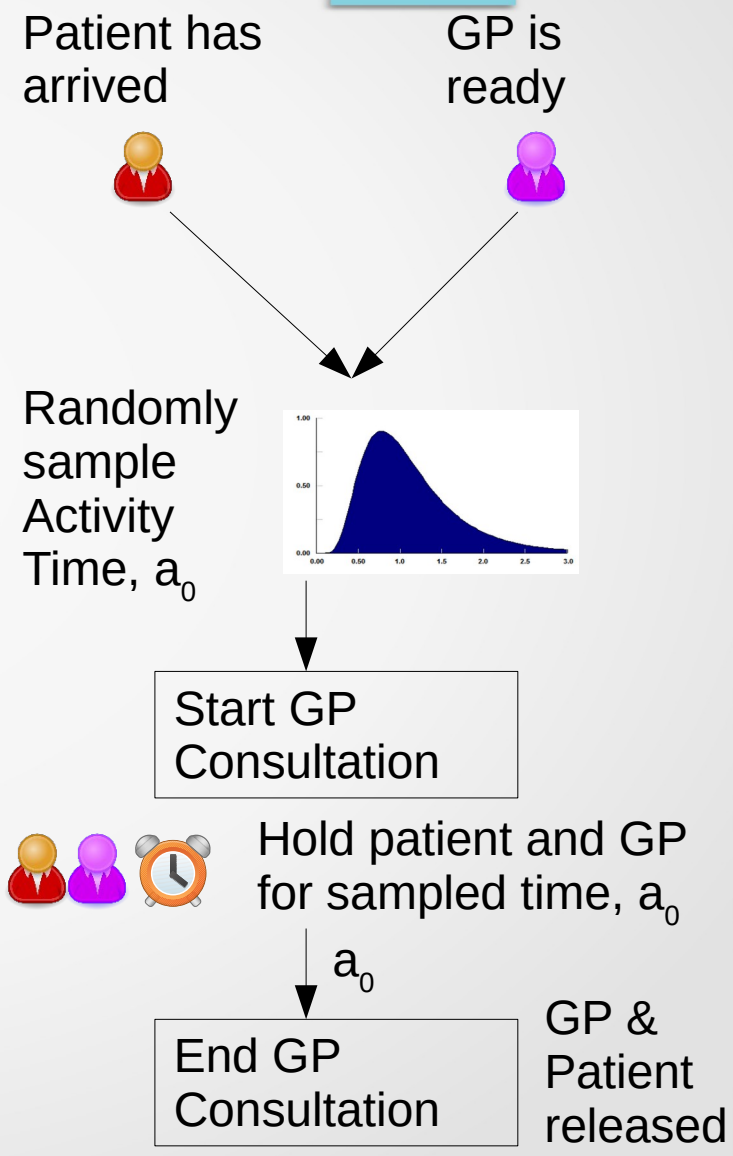
Activities and Activity Times

Each activity in a DES describes a process – this may be a simple atomic task, or a set of tasks bundled together.

For an activity to take place, it needs :
- An entity (drawn from the queue)
- The required type and number of resource to be available

Once the above conditions have been met, the activity begins. The entity, and the resource(s) are then locked in place for an amount of time – the Activity Time. The resource(s) cannot be used elsewhere until the activity time has passed.

Activity times may be fixed, but are typically sampled stochastically from a distribution. Common distributions for process times are Exponential and Log Normal distributions.



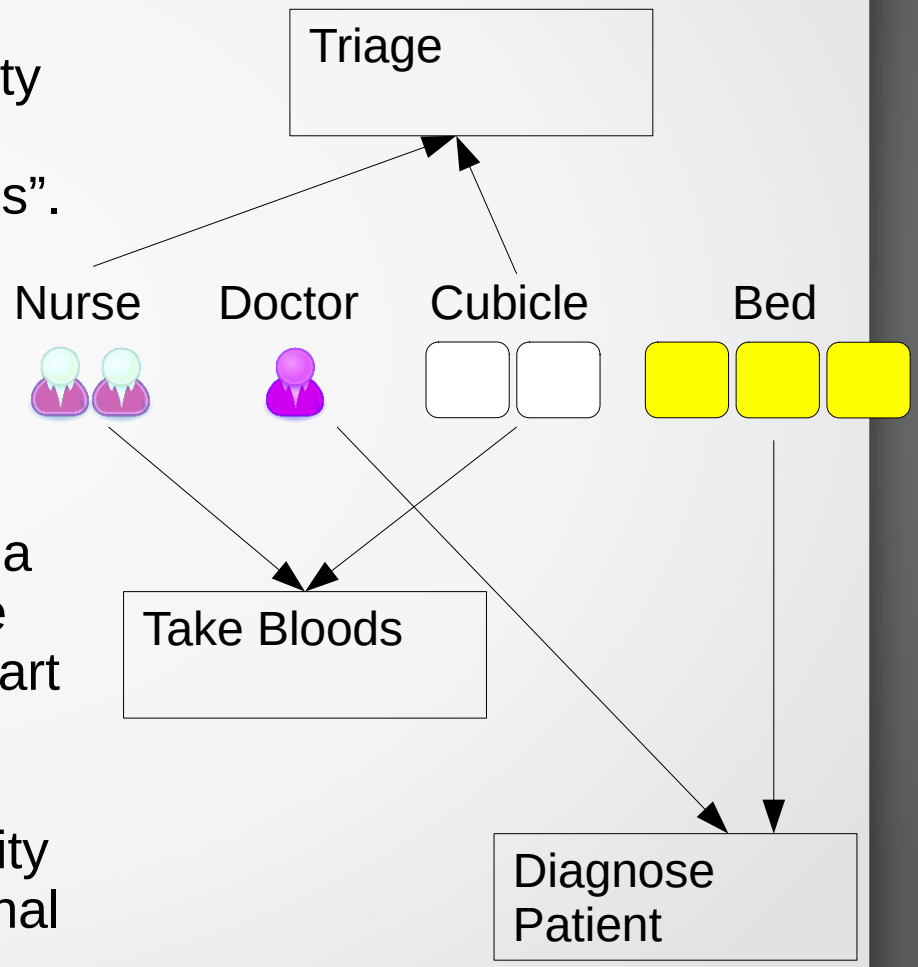
Resources

Resources are needed to undertake activities. An activity may require just a single resource, more than one resource of the same type, or multiple resources of different types. An activity may not require a resource at all, but think carefully to ensure that it really is “resourceless”.

Resources can include “staff” (e.g. doctors, nurses, officers etc) and “stuff” (beds, test equipment, detention cell etc)

Resources can (and often are) shared across a system, so may be required for more than one process. Therefore, a resource drain in one part of the system can affect another.

All required resources are needed for an activity to take place. In some activities, having optional additional resource may speed up the activity (though rarely linearly).



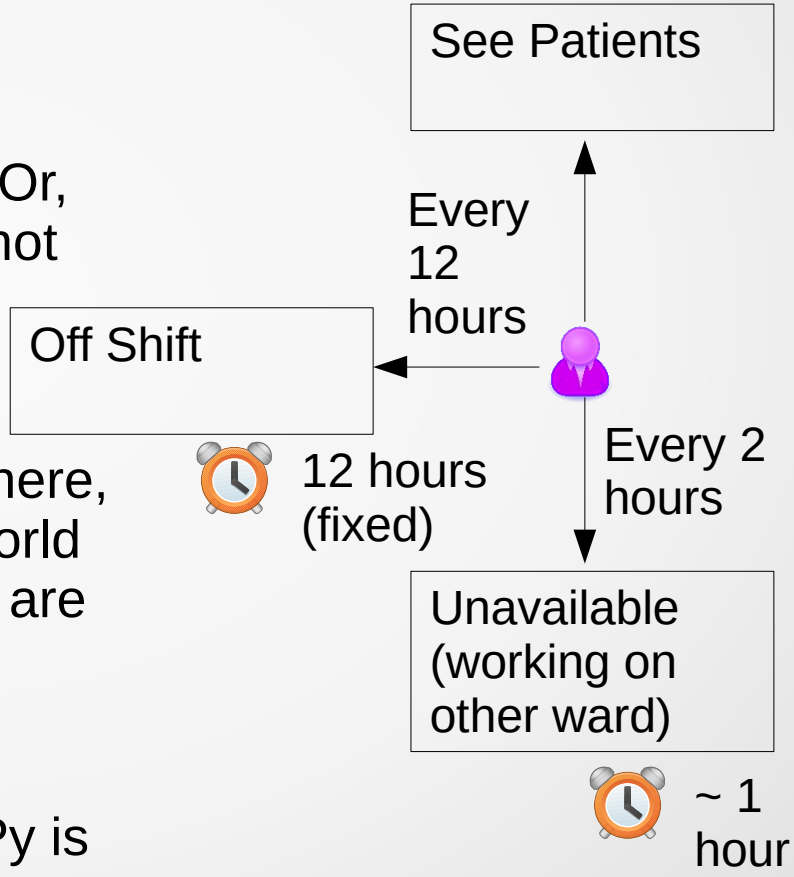
Resource Availability

Resources may be unavailable for an activity because they're working on another activity in the model. But they also may be unavailable for other reasons.

For example, they may be needed in other areas or systems that we're not modelling. Or, they may be unavailable because they are not working / "on shift".

As a modeller, you need to determine what happens when required resources are not there, by working with those involved in the real world system. Do entities wait until the resources are back? Is it not a problem because another resource replaces them? etc

An easy way to model unavailability in SimPy is to have a priority process that grabs the resource you want to make unavailable, and keeps hold of it for a length of time.



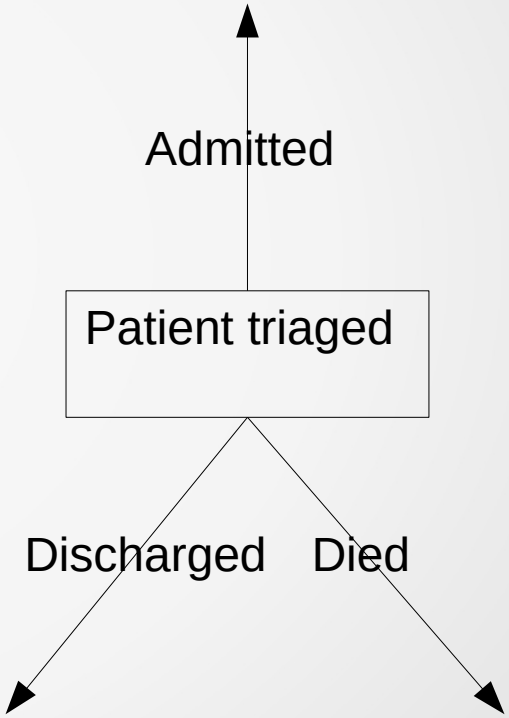
Sinks

Sinks are how entities leave the system, or part of the system, being modelled.

Sinks might include :
- an entity physically leaving a system (e.g. discharge from hospital)
- an entity no longer existing (e.g. death, use of sample)
- an entity no longer needing to access activities that we're interested in

The most important thing to remember about a sink is that it doesn't necessarily represent an entity leaving the system entirely.

For example, the scope of your model may only cover the triage aspect of an Emergency Department. Therefore, a valid sink might be placed after their triage - they've left the scope of our model.

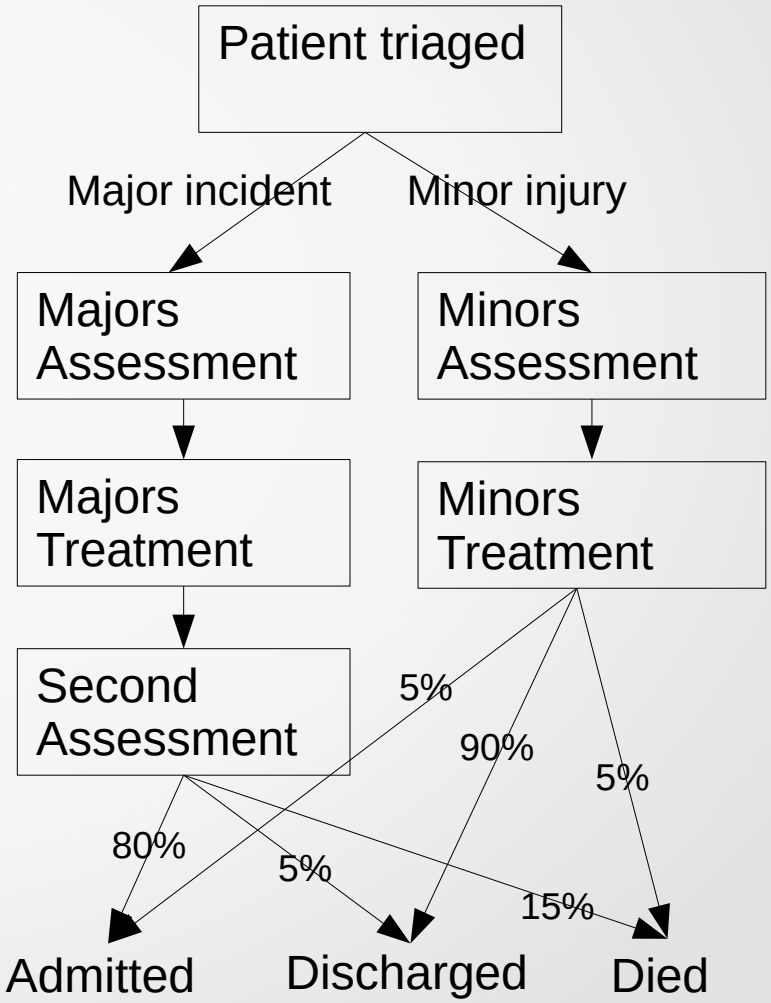


Branching Paths

Real world systems (and the models of those systems) are rarely linear. Often, different things will happen to different entities. In a Discrete Event Simulation, this means different entities flowing to different activities, or different sinks.

We might differentiate based on :

- an attribute of the patient (e.g. patients with a higher priority value flow through a different set of activities)
- probability (e.g. we know that approx 60% of these patients end up being admitted, so we'll randomly select for them to be admitted 60% of the time)
- time (e.g. after a certain time of day, entities flow through a different set of activities)

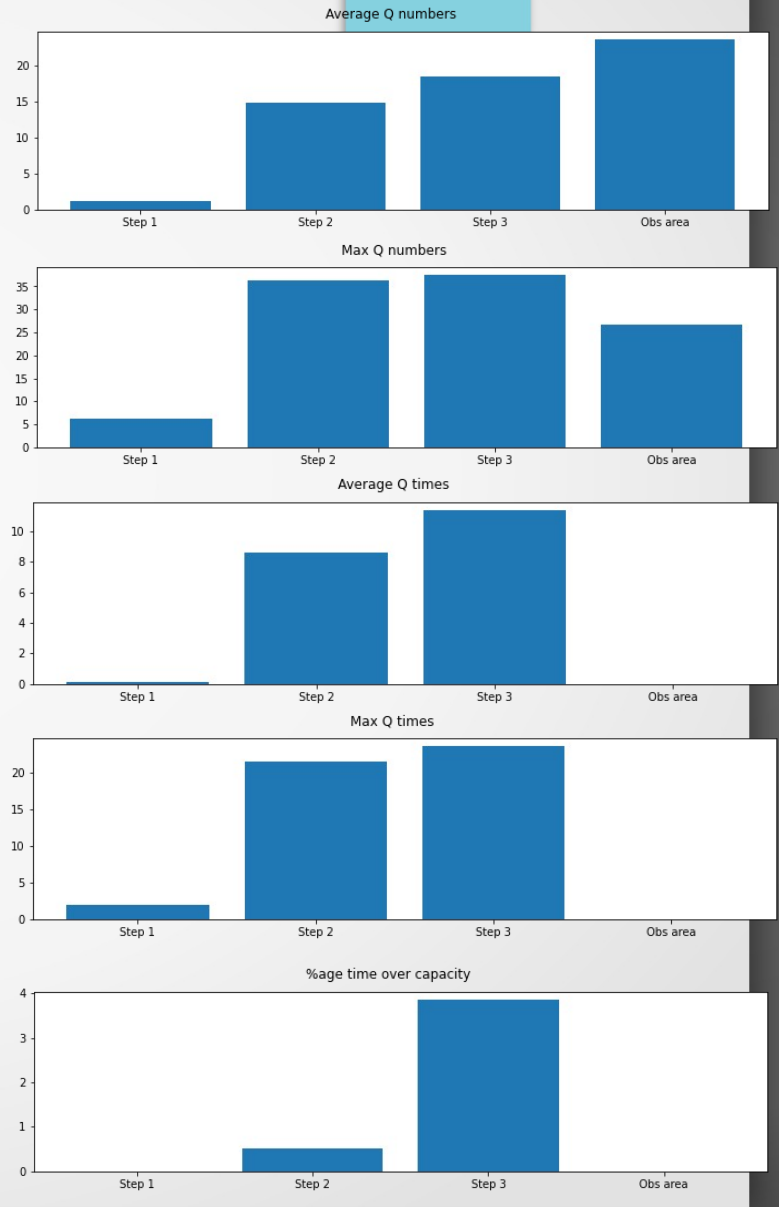


Outputs

As with any type of model, it's important to think about what outputs you need your DES model to generate to answer your modelling questions.

As a DES model is used to model queuing and resourcing problems, typical DES model outputs include average, min, max, xth percentile of :

- time entities are in system
- queue length and duration for queues of interest
- rate of resource utilisation (ie % of time a resource is in use for activities in the model)
- probability of exceeding a defined queue length / queue time / resource utilisation threshold (e.g. 4 hour wait in ED, overcrowding thresholds)



Exercise 1

In your groups, read through the instructions in the file 3a_ex1_instructions.pdf. You are presented with a fictional problem, and you are asked to :

1. Draw a process map of the system described (note that there's more than one process being described here)
2. Write "what if?" question(s) that captures what has been asked of you as the modeller
3. Draw up a design for a Discrete Event Simulation model, based on the "what if?" question(s) that have been asked. Indicate the generators, queues, activities, resources and sinks in the model, and list the entities, inter-arrival times and activity times that will be used in the model. Consider any attributes that your entities might have, and how they might be used. Think about what queuing policies you would have for each queue, and whether you may need to capture reneging, balking or jockeying queuing behaviours. You should also think about what outputs you would want your model to generate to answer your modelling questions. Also, remember our discussion of "scope" from the first session – you may not need to include everything from the process map of the system in your DES model design...

You have 1 hour 15 minutes to complete this exercise. But first take a 5 minute comfort break.

SimPy

SimPy is a Python library that provides a framework for building Discrete Event Simulations.

It has already defined the concepts of entities, activities, inter-arrival times etc so that we don't have to write these things from scratch.

SimPy is built around something in Python called *Generator Functions*. Let's talk about what these are.

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)

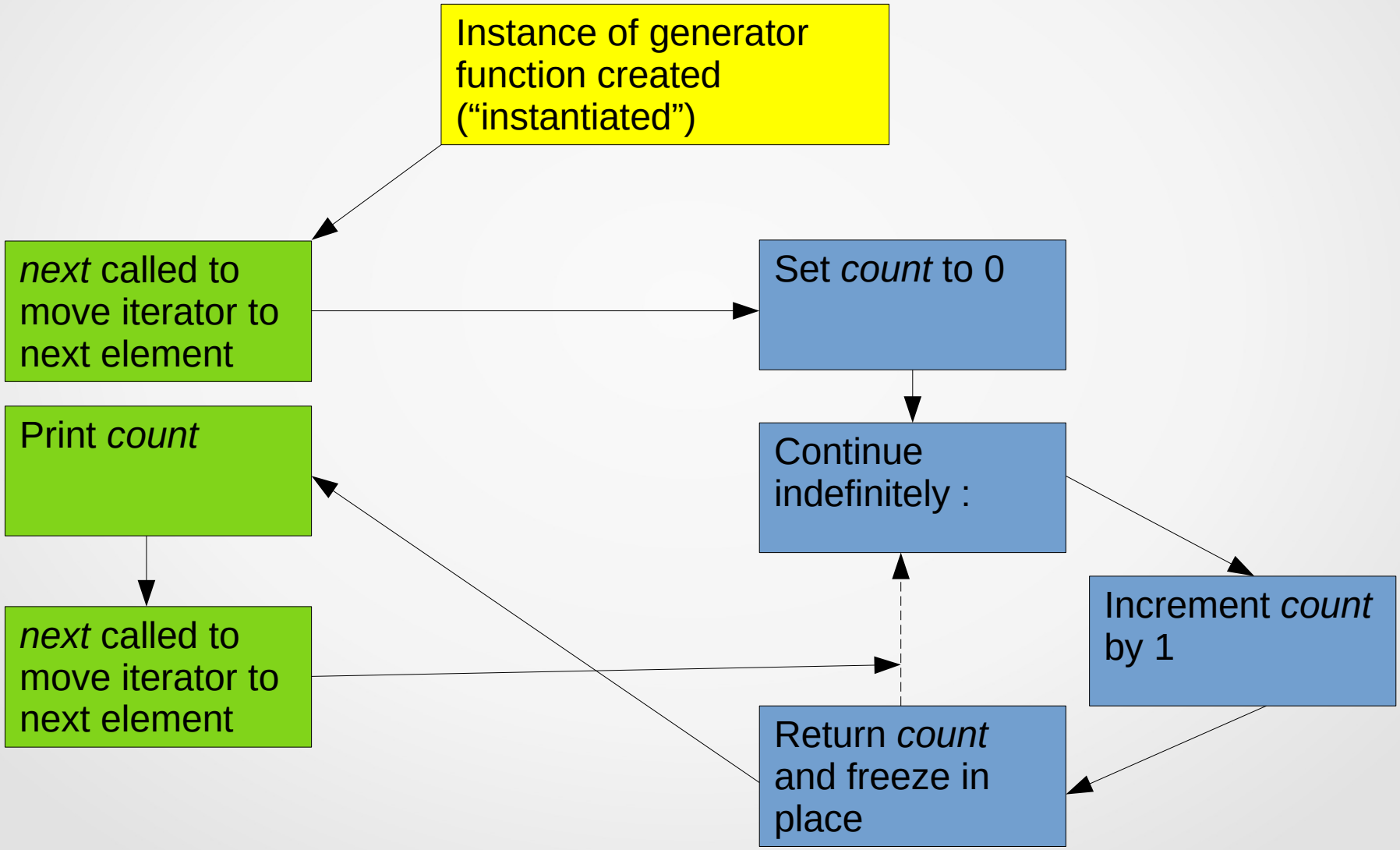
Let’s look at a very simple example of a generator function to see how they work.

```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.  This also means we can have multiple
# instances of the same generator function, all in different "states"
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



Generator Functions in Action in SimPy

Instance of Generator Function to represent our DES Generator (arrivals of entities)

1. Create an instance of the generator function representing the entity's "journey" through the system

2. Start the instance running

3. Sample the time to the next entity arrival

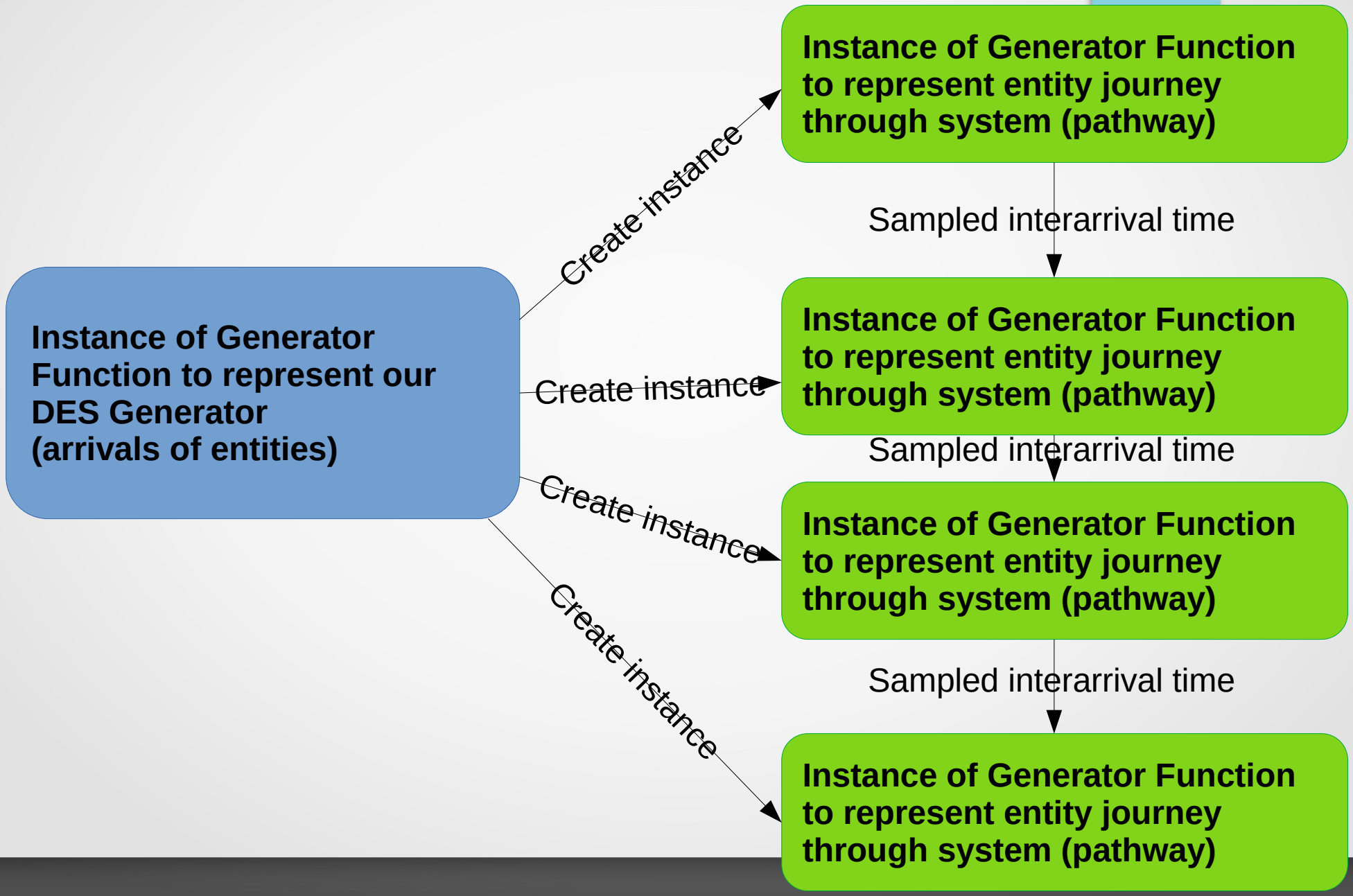
4. Freeze in place using a yield until this time has elapsed

5. Repeat from 1 until end of simulation

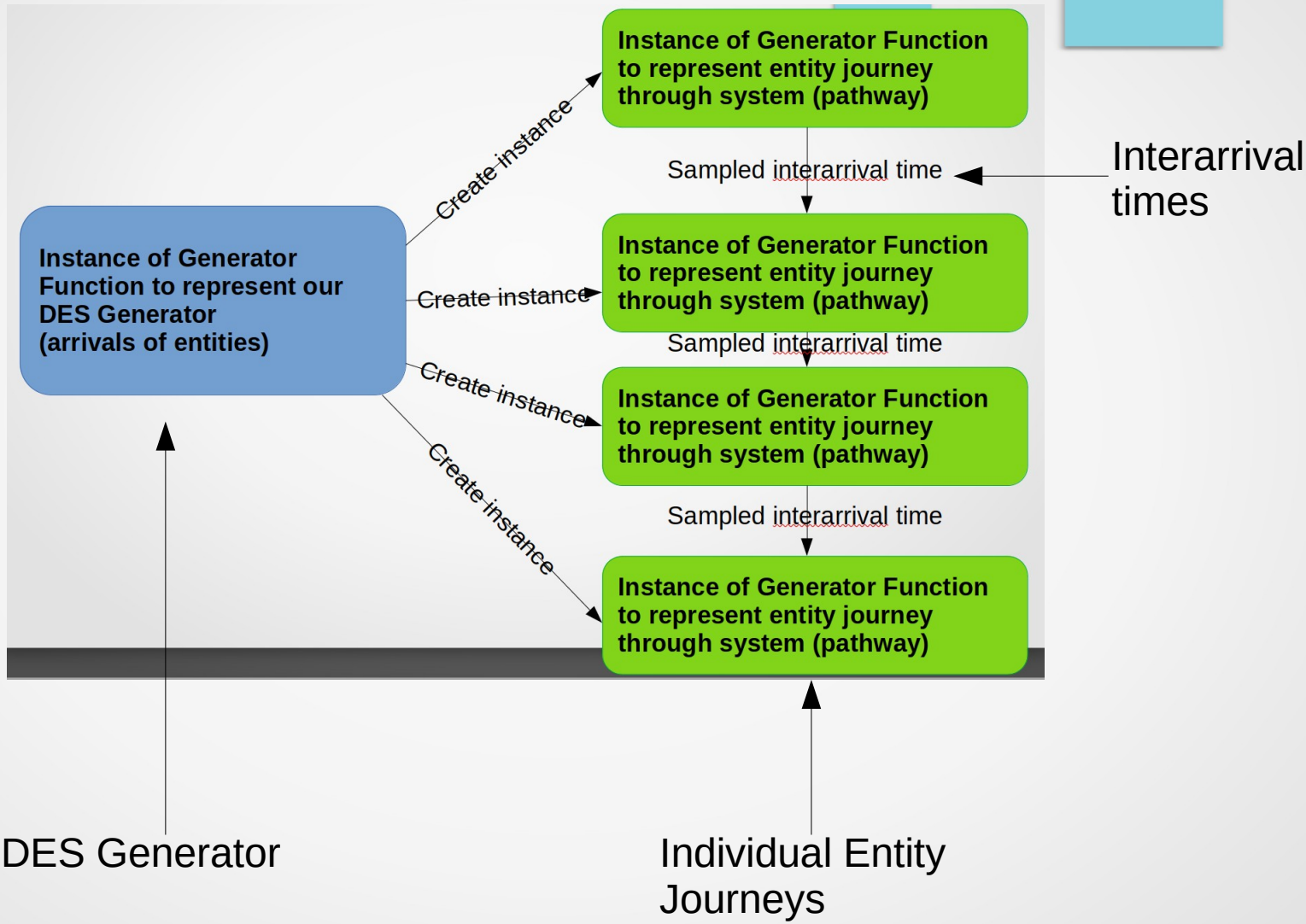
Instance of Generator Function to represent entity journey through system (pathway)

1. Grab time started queuing for activity #1
2. Request required resources
3. Freeze until resources available
4. Grab time finished queuing for activity #1, and calculate queuing time
5. Sample the activity time
6. Freeze in place (holding the resources in place too) for sampled activity time
7. Move to next activity and return to 1. If no more activities, end.

Generator Functions in Action in SimPy



Generator Functions in Action in SimPy

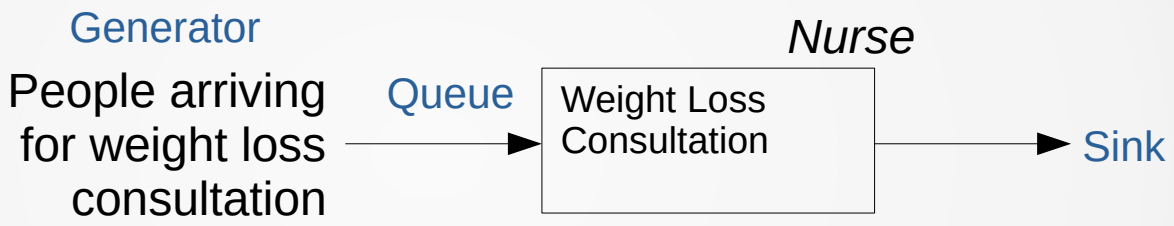


A very simple SimPy example

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

Entities :
- WL Clinic Patients

Activity Times :
- Time spent in weight loss consultation



Let’s look at the code for this. Let’s open simple_simpy.py in Spyder