Bank Example: Understanding Simpy

I have developed a simple simulation modelling a bank that follows roughly the same process that my airtanker simulation model follows for its simulation. By removing lots of the additional complexities that are in that model, I hope to give you a better understanding of the role simpy plays in my simulation model, and help give you an idea of my thought process when creating simulation models with simpy.

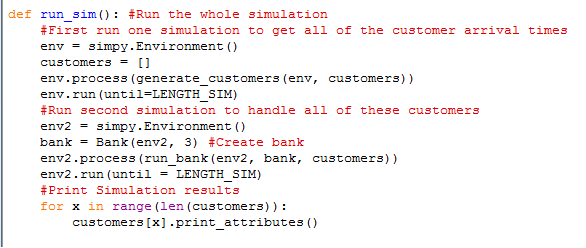
Classes

When using SImpy, the first step is to create classes (objects) that will represent the “pieces” in your simulation. You should choose things that you need to keep track of values for. It is important to try and not have just one “super-class” that holds the parameters and information for everything, as well as you don’t want a bunch of classes each holding just one variable each, or else passing values around in your functions will be confusing and messy. For this simulation I chose to make 3 different classes.

* The Customer. This class will hold all of the important data for one customer in the simulation. I can then make a list of these classes to represent every customer in the whole simulation. The values I considered to be important to keep track of are the times in the simulation the customer arrives at the bank, starts their service time, and finish their service time, as well as an ID for the customer and the counter that served them. Its parallel would be a fire.
* The Counter. This class keeps track of all of the values pertaining to each individual counter in the simulation. Technically since this simulation example is so simple, this class is rather redundant, but I wanted to show how a bank can have a list counter classes that have each of the individual counters’ information, and a list of counter resources for use in the simulation pertaining to queueing. This class keeps track of the counter number, and the customer starting and ending service times in the simulation. Its parallel would be an airtanker.
* The Bank. This class holds all of the information for everything for the bank. In this simulation that only is the list of counter classes for each counter the bank has, as well as the a list of simpy resources for each of these counters that represents each of the counters for use in simpy. The parallel of this would be an airtanker base.

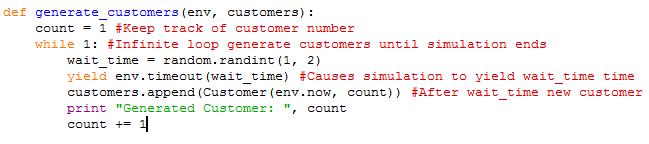
These classes will be used during the whole simulation to run store the simulations information. Now we must create the functions and processes that will be used to run the simulation.

Functions

The first function is the main function to run the simulation. 

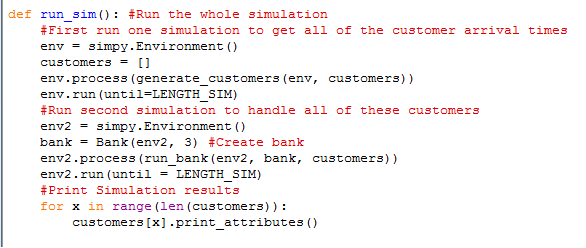
gdgdg

This function is responsible for running the whole simulation when called. First it creates a simpy Environment that will be saved to a variable called env, and an empty list that will hold all of the customers generated I next create a tell simpy to run my generate\_customers function in the simulation. This function will be responsible for creating the customer arrival times. Now that I have set everything up, env.run(until=LENGTH\_SIM) will run all of the processes I have set up until the LENGTH\_SIM time (which is a global variable I can set to any amount of time). I will now go through the generate customers function.



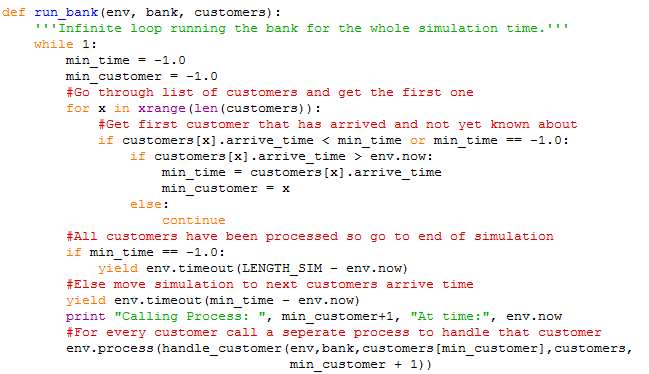
This function is responsible for generating the customers in the simulation. The two arguments are the simpy environment so I can keep track of the current simulation time, and the customers list so I can add new customers to that list. This function is an example of a python generator (but I won’t get into detail about that since it is not important, that is just what simpy uses for its processes). This function first declares a variable count that has an initial value of one. This variable will keep track of each customer’s number so that they can all have a unique id. Next the function runs an infinite loop (while 1 always evaluates to True), so that this function can continuously generate customers for the whole duration of the simulation. In this while loop, a random wait time is generated. Here it is an integer of 1 or 2, but this can be changed anything. The function is then told to wait (yield) for this amount of time to pass in the environment. yield env.timeout(wait\_time) After this amount of time passes, I create a new instance of the customer class and append it to my list of customers, and then increase count by one in preparation for the next customer. This whole process repeats itself until the current simulation time passes the length of time we chose to run the simulation for.

Next, the next part of our main function is run.



I have chosen to divide the customer generation and customer handling parts of the simulation into two separate processes for a couple reasons. 1. As long as the handling of customers will not affect the times that new customers come to the bank, there is no downside in doing so. 2. I find it better to separate things that are separate from each other as it is easier to modify things later on, and you make less mistakes. In this example, this splitting up of the process is quite unnecessary, but I feel in the airtanker simulation model it is a good idea and offers no downside.

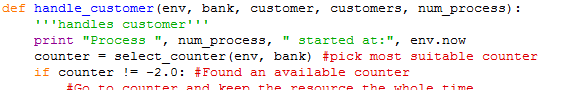
For this second simulation I create a new simulation environment, and create a bank class that will have 3 counters. I then call the run\_bank process in this new environment, and run the environment for the same amount of time. Since Ive done two different environments, I could make the amount of time new customers arrive at the bank, and the amount of time the bank will serve customers for, different values if I wanted to. We will now examine the run\_bank function to see what is going on.



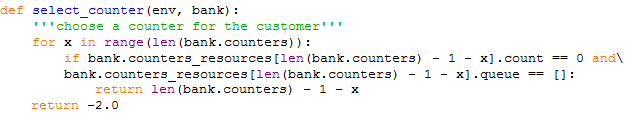
This function is responsible for handling the whole banks operations. Its parameters are the simpy environment so it can keep track of the simulation time, the bank class that holds the counter classes and counter simpy resources, and the list of customers that the bank must service.

This function again runs as an infinite loop so that the function runs for the entire duration of the simulation. Min\_time and min\_customer variables are to keep track of the next customer not yet serviced in the simulation. In this example this is rather redundant since the this follows a FIFO queue, but in the airtanker model this exact process is necessary because the fires follow FIFO based on their report time (and some aren’t reported at all!) not based on their ignition time, so we need to find the next fire based on when it was reported.

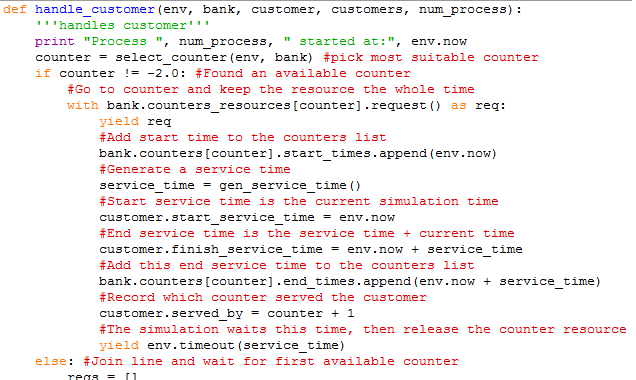
The for loop just goes through the list of customers and finds the one with the earliest arrival time that is larger than the current simulation time (customers with arrival times lower than the current simulation time would already have been handled by previous iterations of this loop). When all of the customers have been handled, the if min\_time == -1.0 will evaluate to True, causing this process to jump to the end of the simulation time (causing the infinite loop to end). Otherwise, the environment keeps skipping ahead in time to the next earliest customers arrival. the yield env.timeout(min\_time – env.now) tells the simulation to wait an amount of time equal to the customers arrival time minus the current simulation time. A new process is then started to handle this customer. This is repeated for every customer in the simulation, all with their own handle\_customer process to handle each individually. I will now go through how that process handles each customer.



This is the first part of the handle\_customer function. I will get into the rest later. This function takes the following parameters: env to keep track of the current environment time, bank to have access to the banks counters, the specific customer being handled, the list of customers (unused in this simulation, but potentially useful if we wanted to add or modify things) and the processes number (the customer that the bank is dealing with). The first thing this function does after printing something is calling the select\_counter function to try and find a suitable counter for this specific customer. It then will return which counter the customer will use. I will now go through that function.



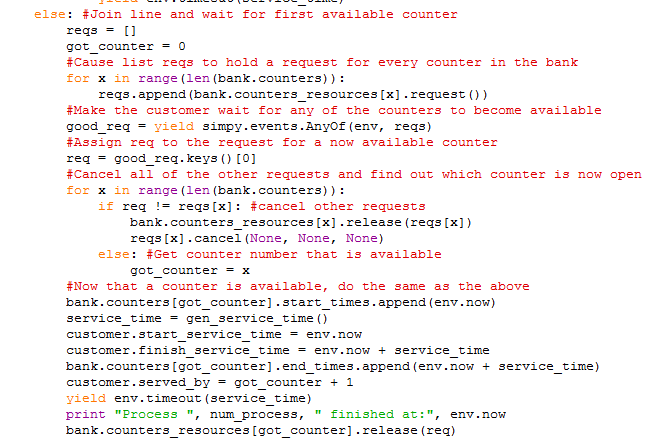
This function selects a counter for the customer to go to. Right now it is just set up to start at the last counter and then go back to the first, if none are available it will return -2.0. This function takes the simpy environment as a variable (not currently needed, but if we wanted to add complexity it would be nice to have), and the bank so we can access its counters and check if they are busy. The loop scans through the list of counters starting at the last one. The count == 0 part of the if statement means nobody is currently using the counter and the queue == [] part means that there is nothing currently in the queue for that specific counter (I actually believe this is a bug with simpy as logically things shouldn’t be in the queue if the counter is available…). If it finds a counter where these conditions are both met, then that counter is available so it returns that counter. If all of the counters are busy it returns -2.0 so the handle costumer function can know that nothing is currently available. I will not go over the handle\_customer function in the case that there is a counter available.



If the counter is not equal to -2.0 then that means a counter was found that is not currently servicing a customer. In this case we use the with line to make this customer have control of the simpy counter resource. The yield line is actually not needed here since we know the counter is available, but this line would normally cause the customer to wait until the counter is free, I have included it in case I modify the code down the road, I wont have to remember to add it if it is later needed. Next, a service time for the customer is generated based on this very simple function. This can obviously be changed to depend on any number of things as long as it returns a time.

C:\Users\Cam\Documents\Airtanker Simulation Model\grunt work\Bank Example\gen_service_time_function.png

The handle\_customer function then records the time that these functions would finish at, before causing the function to hold on to the simpy resource for theservice time in the simulation. Once the function exits this with block, the resource is again available to be used. I chose to do the yield service time part after saving the estimated service time to prevent the simulation time from running out when customers are still being served. The way it is currently set up, all of the customers who get to a counter before the end of the simulation will be served. Any who are in the bank but have not reached a counter by the end of the simulation are considered not served. Next I will go over the other case where the select\_counter function was not able to find an available counter (they are all already in use)



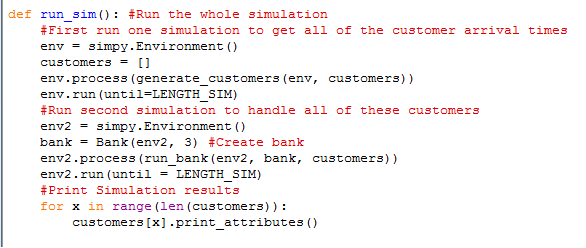
This part is needed if we want the customer to join the first available counter. If we already know which counter we want the customer to join, even if already has a queue, we just use do the same thing as last time. This part may be quite confusing, so just try to bear with me.

We go into the else part of the handle\_customer function because no counter was found to be free. First we create 2 variables, a list called reqs, and a got\_counter variable which we will save the number of the first available counter to. The next for loop causes the customer to request to use every single counter in the bank, and these requests are saved in the reqs list. The function is then told to wait for any of these requests to be fulfilled (ie. At least one counter becomes available). When at least one of the counters is free, all of the requests that passed (all of the requests for counters now free) are saved as a dictionary to the good\_req variable. The next line simply extracts a random\* counter from this (this only matters if two or more counters become available at the exact same simulation time) and saves this request to the variable req.

Now that we have a request that has gone through, we need to loop through all of the counters and cancel all of the other requests (If we already got one counter, why would we need to still go to the others?). Note that it is important to also add the release line here, this prevents us from accidentally taking up two counters if they were both given to us at the same time. The else clause in the for loop means our request matches the request that passed, so that identifies the counter we have access to.

The next steps are identical to the previous ones steps, just get a service time, save the values and then yield the time. The only difference is we now have to release our use of the resource (The other time we didn’t have to since we used a with block, so it was done automatically when the block exited.

These process will run for the entire length of the simulation. When the simulation time passes the length of time we set for it, we can finally finish our main function!



The only thing that is left for us to do is loop through the list of customers and call their print\_attributes() method. This will print the arrive, start service and end of service times for each customer as well as which counter served them. We could also do some statistical work on these values if we wanted to, or run this run\_sim function many times to rerun the simulation.

This example is to serve as a simpler example of the fire model I have developed. Through this document I have tried to show you how simpy works, some of the things we can do with simpy, my personal coding style with simpy, and the general process that the fire model I’ve developed has taken by keeping the exact same process it uses, while removing much of the complexity. I hope you have found that I have at least partially succeeded in these goals.