**Introduction to Simpy (simulation py)-05**

[[Gaurav Kumar](https://gaurav-adarshi.medium.com/?source=post_page-----a0d58f5f40d2--------------------------------)](https://gaurav-adarshi.medium.com/?source=post_page-----a0d58f5f40d2--------------------------------)

[Gaurav Kumar](https://gaurav-adarshi.medium.com/?source=post_page-----a0d58f5f40d2--------------------------------)

·

[Follow](https://medium.com/m/signin?actionUrl=https%3A%2F%2Fmedium.com%2F_%2Fsubscribe%2Fuser%2Ff319c9e06fc1&operation=register&redirect=https%3A%2F%2Fgaurav-adarshi.medium.com%2Fintroduction-to-simpy-simulation-py-05-a0d58f5f40d2&user=Gaurav+Kumar&userId=f319c9e06fc1&source=post_page-f319c9e06fc1----a0d58f5f40d2---------------------post_header-----------)

14 min read

·

Nov 17, 2023

134

3

This is the 5th and final part of the series where we will be discussing how do we actually simulate a real world problem and deduce results from the events. In the last 4 articles we have discussed how we can use simpy to simplify mathematical expressions, define and plot functions and basics of simpy.

**How to Simulate With the**simpy**Package**

In SimPy, the initial step towards conducting a simulation involves selecting a process to replicate. Simulation entails constructing a digital environment that mirrors a real-world system. In this same context, you’ll simulate a scenario for your simulation!

Picture yourself as someone hired to assist the manager of a small local cinema. The cinema has been garnering negative reviews due to prolonged wait times. The manager, equally concerned about expenses and customer satisfaction, can only afford to maintain a limited staff.

The manager is particularly anxious about potential chaos during the release of blockbuster movies: long lines, overworked staff, and frustrated moviegoers missing the beginning of the films. It’s a situation they want to avoid.

After reviewing the feedback, the manager discovered that a typical moviegoer at their cinema is only willing to spend a maximum of 10 minutes from arrival to sitting in a seat. In simpler terms, the average wait time for a night at the cinema must be 10 minutes or less. The manager seeks your assistance in finding a solution to ensure customer wait times meet this 10-minute requirement.

**Brainstorming a Simulation Algorithm**

Before writing any code, it’s crucial to plan how your process would function in the real world. This ensures that the code accurately mirrors the actual customer experience. Here’s a breakdown of how you might consider the steps a moviegoer takes when designing your algorithm:

**1. Arrive at the theater, wait in line, and purchase a ticket.**  
**2. Buy a ticket from the box office.  
3. Wait in line to have the ticket checked.  
4. Get the ticket checked by an usher.  
5. Decide whether to join the concession stand line:**  
— If yes, purchase food.  
— If no, proceed to the next step.  
**6. Find and take their seat.**

This is a detailed sequence for a moviegoer buying a ticket at the theater. You can influence certain aspects, such as reducing wait times by having more cashiers. However, some parts, like the initial arrival of customers, are beyond control. While you can estimate, you can’t precisely determine the number or speed of customer arrivals. To address this, it’s best to use available data to estimate an appropriate arrival time.

**Setting Up the Environment**

Ensure your development environment is set up correctly before initiating the simulation. Begin by importing the required packages through explicit import statements placed at the beginning of your file.

import simpy  
import random  
import statistics

To construct a script for the theater manager, utilize the key libraries. Keep in mind the objective: determining the ideal number of staff to achieve an average wait time below 10 minutes. Gather the duration each moviegoer spends reaching their seats. The subsequent action involves creating a list to store these time values.

wait\_times = []

Create a roster that tallies the overall duration each cinema attendee takes to navigate the entire theater, encompassing the period from their arrival to settling in their designated seat. Initialize this roster at the beginning of the document to enable its utilization in any subsequent function you establish.

**Creating the Environment: Class Definition**

Now, let’s continue building the class definition for the movie theater simulation by adding resources and functions for ushers and servers.

import simpy  
import random  
  
class Theater(object):  
 def \_\_init\_\_(self, env, num\_cashiers, num\_servers, num\_ushers):  
 self.env = env  
 self.cashier = simpy.Resource(env, num\_cashiers)  
 self.server = simpy.Resource(env, num\_servers)  
 self.usher = simpy.Resource(env, num\_ushers)  
 def purchase\_ticket(self, moviegoer):  
 yield self.env.timeout(random.randint(1, 3))  
 def sell\_food(self, moviegoer):  
 yield self.env.timeout(random.randint(1, 5))  
 def check\_ticket(self, moviegoer):  
 yield self.env.timeout(3 / 60) # 3 seconds converted to minutes

In the updated class definition, we’ve added two new resources (server and usher) to represent servers who sell food and ushers who check tickets. Each resource is initialized with the corresponding number of servers or ushers available.

We’ve also defined two new functions, sell\_food() and check\_ticket(), to simulate the time it takes for moviegoers to place food orders and receive their food, and for ushers to check tickets, respectively.

The sell\_food() function uses random.randint(1, 5) to generate a random number between 1 and 5, representing the time in minutes it takes for a moviegoer to place an order and receive their food.

The check\_ticket() function uses self.env.timeout(3 / 60) to introduce a time delay of 3 seconds, converted to minutes. This represents the time it takes for ushers to check a moviegoer's ticket.

With these additions, you now have a more comprehensive class definition for the movie theater environment, including resources and functions to simulate ticket purchasing, food ordering, and ticket checking processes.

**Moving Through the Environment: Function Definition**

Alright, so you’ve established your simulation environment with a class defining resources and processes. Now, you need a function to model the movement of moviegoers through this environment. Let’s call this function go\_to\_movies().

def go\_to\_movies(env, moviegoer, theater):  
 # Moviegoer arrives at the theater  
 arrival\_time = env.now

This function takes three arguments:

* env: The environment controlling the moviegoer.
* moviegoer: A variable tracking each person's journey through the system.
* theater: Provides access to the processes defined in the overall class.

The arrival\_time variable stores the time each moviegoer arrives at the theater, obtained using env.now.

Now, for each process in the theater class, there’s a corresponding request in go\_to\_movies():

with theater.cashier.request() as request:  
 yield request  
 yield env.process(theater.purchase\_ticket(moviegoer))

In this example, the first process is purchase\_ticket(), which requires the use of a cashier resource. The moviegoer requests the cashier, waits if it’s currently in use, and then completes the purchase\_ticket() process.

This is achieved through the with statement, yielding the request and the process. The yield request line represents the moviegoer waiting for a cashier to become available, and yield env.process() signifies the actual use of the cashier for purchasing the ticket.

After using a resource, it needs to be freed up for the next user. The with statement implicitly releases the resource once the process is complete.

The same structure is repeated for other processes, like checking the ticket:

with theater.usher.request() as request:  
 yield request  
 yield env.process(theater.check\_ticket(moviegoer))

And optionally, buying food:

if random.choice([True, False]):  
 with theater.server.request() as request:  
 yield request  
 yield env.process(theater.sell\_food(moviegoer))

Here, randomness is introduced to decide whether a moviegoer will buy food or not.

Finally, the function tracks the time the moviegoer finishes all processes and heads into the theater:

# Moviegoer heads into the theater  
 wait\_times.append(env.now - arrival\_time)

The departure time is calculated by subtracting the arrival\_time from the current simulation time (env.now), and the result is appended to the wait\_times list.

This way, the go\_to\_movies() function models the entire journey of a moviegoer through the simulated environment.

**Making Things Happen: Function Definition**

To implement the simulation, you must define a function named run\_theater(). This function will orchestrate the entire simulation by creating a theater instance and generating moviegoers until the simulation concludes. The initial step within this function is to instantiate a theater:

def run\_theater(env, num\_cashiers, num\_servers, num\_ushers):  
 theater = Theater(env, num\_cashiers, num\_servers, num\_ushers)

Since this function serves as the main process, it is crucial to pass all previously declared variables, namely num\_cashiers, num\_servers, and num\_ushers, to the function. This ensures that the simulation has control over these essential variables. The next step involves defining a variable named theater and initializing the theater with the specified number of cashiers, servers, and ushers.

To simulate an initial group of moviegoers waiting at the theater, you can anticipate a few individuals eager to buy tickets as soon as the box office opens. You can accomplish this by adding the following code snippet:

for moviegoer in range(3):  
 env.process(go\_to\_movies(env, moviegoer, theater))

Here, the range() function populates the theater with three moviegoers, and the env.process() function instructs the simulation to prepare for moving them through the theater. The remaining moviegoers will arrive at the theater at their own pace.

Since the arrival time of new moviegoers is unknown, you decide to base it on past data. After analyzing timestamped receipts, you discover that, on average, moviegoers arrive every 12 seconds. To incorporate this into the function, introduce a waiting period before generating a new person:

while True:  
 yield env.timeout(0.20) # Wait a bit before generating a new person

Note that the decimal number 0.20 represents 12 seconds, calculated by dividing 12 seconds by 60 seconds (the number of seconds in a minute).

Following this waiting period, increment the moviegoer variable by 1 and generate the next person using the same generator function used for the initial three moviegoers:

moviegoer += 1  
 env.process(go\_to\_movies(env, moviegoer, theater))

This completes the function. When you call run\_theater(), the simulation will commence with three initial moviegoers, progressing through the theater with the go\_to\_movies() function. Subsequently, new moviegoers will arrive at 12-second intervals and move through the theater at their own pace.

**Calculating the Wait Time: Function Definition**

To compute the average wait time for moviegoers, you’ll first need a list, wait\_times, containing the total time each individual spent from arrival to ticket check. Let’s define a function, get\_average\_wait\_time(), to handle this task:

import statistics  
  
def get\_average\_wait\_time(wait\_times):  
 average\_wait = statistics.mean(wait\_times)

This function accepts the wait\_times list as an argument and employs the statistics.mean() method to determine the average wait time.

Now, for better user readability, it’s beneficial to create another function, calculate\_wait\_time(), specifically tailored for the movie theater manager:

def calculate\_wait\_time(arrival\_times, departure\_times):  
 average\_wait = statistics.mean(wait\_times)  
   
 # Pretty print the results  
 minutes, frac\_minutes = divmod(average\_wait, 1)  
 seconds = frac\_minutes \* 60  
   
 return round(minutes), round(seconds)

Within this function, the divmod() function is utilized to present the results in both minutes and seconds. This ensures that the output is user-friendly and easily comprehensible for the theater manager.

**Choosing Parameters: User Input Function Definition**

In the process of developing simulation functions, you encounter certain variables that lack explicit definitions, namely, num\_cashiers, num\_servers, and num\_ushers. These variables serve as parameters, offering flexibility to observe how the simulation behaves under different conditions. For instance, when dealing with a high-demand movie, adjusting the num\_cashiers parameter allows you to explore the impact on customer service. Similarly, tweaking the num\_servers parameter helps optimize the flow, especially at the concessions stand.

Simulation, at its core, empowers you to experiment and make informed decisions by simulating real-life scenarios.

To facilitate user interaction and customization, you’ll design a helper function named get\_user\_input():

def get\_user\_input():  
 num\_cashiers = input("Input # of cashiers working: ")  
 num\_servers = input("Input # of servers working: ")  
 num\_ushers = input("Input # of ushers working: ")  
 params = [num\_cashiers, num\_servers, num\_ushers]  
   
 if all(str(i).isdigit() for i in params): # Ensure input validity  
 params = [int(x) for x in params]  
 else:  
 print(  
 "Could not parse input. The simulation will use default values:",  
 "\n1 cashier, 1 server, 1 usher."  
 )  
 params = [1, 1, 1] # Default values  
   
 return params

This function leverages Python’s input() function to gather user input. Since user input can be unpredictable, an if/else clause is employed to handle potential invalid entries. If the input is deemed invalid, default values (1 cashier, 1 server, 1 usher) are set for the simulation to proceed smoothly. This ensures a seamless and user-friendly experience when experimenting with different parameter values.

**Finalizing the Setup: Main Function Definition**

Now, let’s define the main() function to ensure the proper execution order of your script. The main() function orchestrates the entire process when you run your script from the command line. Here’s a step-by-step guide along with a coding example:

**Create the main() Function:**

def main():

**Setup: Initialize the random seed to maintain result consistency.**

random.seed(42)

**User Input: Get input from the user using the get\_user\_input() function.**

num\_cashiers, num\_servers, num\_ushers = get\_user\_input()

**Simulation Execution: Create the simulation environment using SimPy.**

env = simpy.Environment()

**Run the theater simulation process with the specified parameters.**

env.process(run\_theater(env, num\_cashiers, num\_servers, num\_ushers))

1. **Simulation Duration:**Set the simulation duration using **env.run(until=90)** to simulate 90 minutes.
2. **Results Retrieval:**Get and store the average wait time using the get\_average\_wait\_time() function.

mins, secs = get\_average\_wait\_time(wait\_times)

**Display Results: Print the simulation results for the user to view.**

print("Running simulation...", f"\nThe average wait time is {mins} minutes and {secs} seconds.")

**Explanation:**

* Initialize the random seed for consistent results.
* Collect user input for simulation parameters.
* Create the simulation environment and run the theater simulation.
* Specify the simulation duration.
* Retrieve and display the average wait time.

By implementing these steps, your main() function is now ready to execute the entire simulation process, providing meaningful results to the user. With this, your setup is complete!

**How Simulation Works?**

Simulation in Python is achieved using the simpy framework, and it involves orchestrating a model of a system to imitate real-world processes. Here’s a concise explanation with a code example:

1. **Environment Setup:** To initiate a simulation, you create an environment using simpy. It’s like setting up a virtual space where your simulation will unfold.

import simpy

env = simpy.Environment()

1. **Parameter Definitions:** Define the parameters that affect the simulation. These parameters represent elements you want to study, like the number of booths, the time for ID checks, and passenger arrival rates.

num\_booths = 5  
check\_time = 3  
passenger\_arrival = 2

1. **Simulation Execution:** You’ll define the simulation’s logic, typically as functions, and then use env.process() to schedule them within the environment. Finally, start the simulation with env.run(until=sim\_time).

def checkpoint\_run(env, num\_booths, check\_time, passenger\_arrival):  
 # Define your simulation logic here  
 env.process(checkpoint\_run(env, num\_booths, check\_time, passenger\_arrival))  
 env.run(until=10)

To sum it up, simulating a system in Python involves creating an environment, specifying parameters, and executing the simulation by scheduling events within that environment. The simulation unfolds virtually and quickly, despite the real-time units used for reference.

**How to Run the Simulation?**

With just a few additional lines of code, you’ll be poised to witness your simulation spring to action. Before that, here’s a summary of the functions and classes you’ve established thus far:

* **Theater**: This class description functions as a template for the setting you wish to replicate. It specifies various details about the given setting, such as the available assets and associated activities.
* **go\_to\_movies():**This function involves making clear and direct appeals to access a resource, following the associated procedure, and subsequently relinquishing it for the next individual in line at the cinema.
* **run\_theater():**This function is responsible for overseeing the simulation. It starts by instantiating a theater using the Theater class blueprint and proceeds to invoke the go\_to\_movies() method, which orchestrates the creation and movement of individuals within the theater.
* **get\_average\_wait\_time():**This function calculates the mean duration required for a cinema attendee to navigate through the movie theater.
* **calculate\_wait\_time():**This function guarantees that the ultimate result is presented in a user-friendly and comprehensible format.
* **get\_user\_input():**This function empowers the user to specify various parameters, such as the number of available cashiers.
* **main():**This function guarantees that your script operates smoothly when executed in the command line.

You just need to add two more lines of code to call your main function:

if \_\_name\_\_ == '\_\_main\_\_':  
 main()

With those instructions, your script is prepared for execution! Open your terminal, go to the directory where you’ve saved simulate.py, and execute the subsequent command:

$ python simulate.py  
Input # of cashiers working:

You will be allowed to choose the parameters for your simulation. Below, you can see an example of the output generated using the default settings:

$ python simulate.py  
Input # of cashiers working: 1  
Input # of servers working: 1  
Input # of ushers working: 1  
Running simulation...  
The average wait time is 42 minutes and 53 seconds.

That’s quite an extended period to linger!

**When to Change Things Up**

Remember, the key is to present a solution to the manager regarding the optimal number of employees to maintain wait times under 10 minutes. Start by experimenting with various parameters to identify the most effective numbers.

**Initial Exploration:** Begin with an extreme scenario by maximizing the resources. Hypothetically, employ 100 cashiers, 100 servers, and 100 ushers — even though this is impractical. The aim is to quickly determine the system’s limits.

python simulate.py   
Input # of cashiers working: 100   
Input # of servers working: 100   
Input # of ushers working: 100   
Running simulation... The average wait time is 3 minutes and 29 seconds.

1. Even with maximum resources, the wait times remain above the desired 10-minute threshold. Proceed to adjust the numbers to achieve the manager’s target.
2. Optimization Attempt: Refine the parameters to reduce wait times to the specified 10 minutes. Experiment with different combinations until you find a plausible solution. Below is one possible adjustment:

python simulate.py   
Input # of cashiers working: 20   
Input # of servers working: 30   
Input # of ushers working: 15   
Running simulation... The average wait time is 9 minutes and 57 seconds.

3. After identifying a solution, present the results to the manager and propose improvements, such as installing ticket kiosks to cut costs.

**Conclusion**

Throughout this tutorial, you’ve gained insights into building and running simulations using Python’s simpy framework. Understanding how systems operate and creating virtual representations to mitigate congestion and delay is crucial.

**Now, equipped with the knowledge to:**

* Brainstorm simulation algorithms step by step
* Establish a virtual environment in Python with simpy
* Define functions representing agents and processes
* Modify simulation parameters for optimal solutions

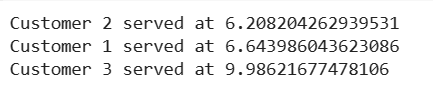
Apply this knowledge to various scenarios. Simpy offers a versatile toolset; continue exploring and optimizing processes to save time and money. Your newfound skills can make a significant impact in diverse applications. Dive in and discover more ways to enhance efficiency!

**Let’s take one more simulation:**

Let’s create a simple event-driven simulation of a bank with customers arriving, waiting in line, and being served by tellers. We’ll use SimPy for this example.

import simpy  
import random

# Step 1: Define the Bank class  
class Bank:  
 def \_\_init\_\_(self, env, num\_tellers):  
 self.env = env  
 self.tellers = simpy.Resource(env, capacity=num\_tellers) def serve\_customer(self, customer):  
 yield self.env.timeout(random.uniform(0.5, 1.5))  
 print(f"Customer {customer} served at {self.env.now}")# Step 2: Define the customer arrival process  
def customer\_arrival(env, bank):  
 customer = 1  
 while True:  
 yield env.timeout(random.expovariate(1.0 / 5)) # Customer arrival follows an exponential distribution  
 env.process(bank.serve\_customer(customer))  
 customer += 1# Step 3: Create the simulation environment  
env = simpy.Environment()# Step 4: Create an instance of the Bank class  
bank = Bank(env, num\_tellers=2)# Step 5: Start the customer arrival process  
env.process(customer\_arrival(env, bank))# Step 6: Run the simulation  
env.run(until=10)



**Output**

1. **Define the Bank class:** Create a class called Bank with a resource representing the tellers. The serve\_customer method simulates the time it takes to serve a customer.
2. **Define the customer arrival process:** Use a generator function customer\_arrival to model customers arriving at the bank. Customers arrive according to an exponential distribution, and each customer is served by the bank.
3. **Create the simulation environment:** Instantiate a SimPy environment.
4. **Create an instance of the Bank class:** Instantiate the Bank class with a specified number of tellers.
5. **Start the customer arrival process:** Begin the simulation by starting the customer arrival process.
6. **Run the simulation:** Use env.run(until=10) to run the simulation for a specified time (in this case, 10 time units).

This example is a simplified simulation, and you can extend it by adding more features, such as tracking customer wait times, introducing additional services, or varying the number of tellers dynamically.

— — — — — — — — — — — — — — — — 5 — — — — — — — — — — — — — — — — — — — —

In the last 5 article series, we have seen how we can use simpy to simulate discreet events and most of its functionality. I really feel that it can really help practitioners from STEM background in their day to day work and more so to decision makers to simulate and see how their decision might impact other loosely held factors.

Let me know in case you have any doubt. I’ll be soon be coming out with a video series for this library. Till then, keep learning and keep enjoying!!! And yes, do follow me for more such Tech Articles.