THE ESSENTIAL GUIDE TO MASTERING SIMULATION IN PYTHON

SIMULATION IN PYTHON WITH SIMPY

A COMPREHENSIVE GUIDE TO BUILDING AND ANALYSING SIMULATIONS WITH REAL-WORLD APPLICATIONS

Harry Munro CEng

Simulation in Python with SimPy

Version 1.2 - Last updated 7th March 2025

Dear Reader,

This guide is designed to provide you with a practical, easy-to-understand overview of building simulations in Python. I will introduce you to SimPy, a powerful Python library for discrete-event simulations. Whether you're new to simulation in Python or looking for a concise reference, this document covers the essentials to help you build powerful simulations.

Inside, you'll find:

- Clear explanations of key SimPy components like events, processes, and resources
- Step-by-step examples to help you simulate real-world systems
- Best practices for writing clean, scalable simulations
- Tips for managing queues, resources, and handling time in your models

Use this guide as a quick reference guide when building simulations, whether you're tackling a simple task or a complex system. If you're ready to dive deeper, this is just the beginning - there's much more to explore in the world of simulation..!

Yours sincerely,

Harry Munro
Founder, TeachEm

School
of
Simulation

Table of Contents

School of Simulation

School of Simulation

School of Simulation

School of Simulation

School of Simulation

Introduction

Problems on the London Underground

A few years into my engineering career, I was handed a challenge that would redefine my approach to simulation and set the course for my professional journey. The project involved building probabilistic simulations for the London Underground – a complex, high-stakes task that required predicting the capacity of various sites and enabling the network to forecast how future projects would perform.

At the time, there was an off-the-shelf simulation tool available. It was the obvious choice – expensive, proprietary, and limited. It worked well enough for standard problems but struggled with the nuances of the Underground's unique systems. The costs and constraints didn't sit well with me, so I began exploring alternatives.

That's when I discovered Python and its SimPy library. SimPy is a lightweight tool designed for discrete-event simulation, and its potential was immediately clear. It was open-source, flexible, and free of the licensing fees that came with traditional software. Despite having no prior experience with Python, I was intrigued by the possibilities it offered.

I proposed using SimPy to develop the simulations. This wasn't an easy sell – stakeholders were naturally cautious about moving away from tried-and-tested software to an open-source solution. But I believed in the value it could bring, so I set out to prove its worth. The transition was not without its hurdles. I was learning Python as I went, figuring out version control for the first time, and navigating the complexities of creating detailed simulations from scratch. Yet, the more I worked with SimPy, the more its advantages became apparent. It allowed me to model complex sites on the Underground network – from depots to intricate junctions – with a level of detail and customisation that the proprietary tool couldn't match.

The results were transformative. The simulations we built with SimPy provided insights that shaped multimillion-pound decisions. They gave the Underground the ability to predict outcomes with confidence, optimise operations, and plan for the future. The success of this project didn't just validate SimPy – it solidified Python as the foundation for my team's simulation work moving forward.

This experience taught me more than just how to use Python for simulation. It showed me the power of challenging the status quo, embracing new tools, and trusting in my ability to learn and adapt. These lessons have stayed with me, and I hope they'll inspire you as you embark on your own journey into simulation with Python.

Introducing Discrete-Event Simulation (DES)

Imagine conducting an orchestra - but instead of continuous music, each instrument plays only at precisely the right moment. Violins strike as the curtain rises, flutes enter exactly at the crescendo, and the timpani crashes perfectly on cue. It's less about constant harmony, more about impeccable timing. In fact, it’s a bit like managing my toddler's tantrums - timing really is everything.

Discrete-Event Simulation (DES) brings this idea of precision and timing into the world of systems and processes. Rather than looking at smooth, continuous operations, DES focuses on key events that drive performance. Whether you're fine-tuning a bustling factory floor, managing patient flow in an overcrowded hospital, or handling bursts of network traffic, DES lets you zoom in on those critical moments where everything either clicks into place or falls spectacularly apart.

With DES, you can tackle the big 'what if' questions without the guesswork: What if a crucial machine stops working? How would hiring extra hospital staff impact patient wait times? Could the network actually survive an unexpected surge in traffic, or would it collapse faster than my diet on holiday?

Instead of relying on intuition or luck, DES provides clear, actionable insights grounded in realistic scenarios - helping you make smarter decisions, faster.

With such capabilities, it’s no surprise that DES is an essential tool for engineers, scientists, and analysts working in today’s complex, data-driven world. But as we’ll explore, not all simulation approaches are created equal - and that’s where Python, and its powerhouse library SimPy, enter the stage.

The Importance of Learning from Simulation

Simulation offers one of the greatest benefits in engineering and technology: the freedom to make mistakes. As John H. McLeod wisely noted, in real life, mistakes can be costly and irreversible. However, with computer simulations, you can afford to make mistakes on purpose, learn from them, and iterate without facing the real-world consequences.

This is where simulation becomes truly invaluable. It allows us to explore scenarios, test different strategies, and fail safely. Each failure provides new insights, refining our understanding and leading us to better solutions. Moreover, for large capital-intensive projects, simulation offers immense cost-saving opportunities by allowing more optimal decision-making early in the project lifecycle. By catching potential issues or inefficiencies before they occur in reality, simulation can reduce waste, improve resource allocation, and avoid costly redesigns or delays.

I encourage you, as the reader, to embrace this mindset. Dive into simulations with the intention to experiment, fail, and learn, knowing that each iteration brings you closer to mastering the complexities of the system you're modelling.

The Problem with Proprietary Simulation Software

For years, simulation engineers have relied on proprietary tools to tackle complex challenges. These tools are often marketed as all-encompassing solutions, but they come with significant drawbacks that can stifle innovation and limit their utility in fast-evolving fields.

High Licensing Costs and Constraints

Proprietary software often requires expensive licences, which can be cost-prohibitive for smaller organisations or individuals. These costs aren't a one-time investment either – annual renewal fees quickly add up. For engineers working on tight budgets or freelancers striking out on their own, these costs can create a significant barrier.

Vendor Lock-In

Proprietary tools rarely integrate seamlessly with external systems. They are designed to keep users within their ecosystem, creating dependency on a single vendor. This lock-in reduces flexibility and increases long-term costs, particularly if you need features not included in the software's package or decide to move to a different solution.

Limited Flexibility for Niche or Evolving Applications

Simulation isn't a one-size-fits-all practice. Many proprietary tools are built for general-purpose use, which limits their ability to adapt to unusual or emerging domains. For engineers working in fields like renewable energy, autonomous systems, or highly customised industrial processes, this rigidity can be a critical obstacle.

Opaque, Black-Box Models

Proprietary software often hides the mechanics of its calculations. For engineers, this lack of transparency makes debugging, customisation, and improvement almost impossible. It's a frustrating limitation that hampers creativity and trust in the models being built.

Reduced Collaboration and Reproducibility

Sharing work across teams or organisations becomes difficult when software is proprietary. Others need access to the same tools and licences, making collaboration both expensive and cumbersome. Moreover, the black-box nature of these tools can hinder reproducibility – a cornerstone of engineering and scientific work.

In a field that thrives on innovation and precision, these limitations are significant. Engineers often need to work around the constraints of the software, wasting time and effort that could be better spent solving problems. This is why Python – and SimPy – are rapidly becoming the tools of choice for forward-thinking engineers and analysts.

Why Simulations in Python Are a Game-Changer

When it comes to overcoming the limitations of proprietary software, Python offers a breath of fresh air. With its open-source nature and a vast ecosystem of libraries, Python has become the go-to language for simulation engineers seeking flexibility, transparency, and cost-effectiveness. Here's why simulations in Python, particularly with SimPy, are revolutionising the field:

Complete Flexibility and Control

Unlike proprietary tools, Python offers unmatched adaptability. Whether you're modelling a standard process or a niche system, Python allows you to customise every aspect of your simulation. You're not confined to pre-set templates or limited options – you build exactly what you need, tailored to your specific requirements.

Clean, Intuitive Code with SimPy

SimPy, Python's most popular discrete-event simulation library, is lightweight, easy to learn, and incredibly powerful. It enables you to write clean, understandable code that doesn't feel like a maze of technical jargon. With SimPy, you focus on solving the problem, not wrestling with the tool.

Leverage the Entire Python Ecosystem

Python's ecosystem is one of its greatest strengths. You can seamlessly integrate your simulations with data science libraries like pandas and NumPy, visualisation tools like Seaborn, and machine learning frameworks such as TensorFlow. This connectivity allows you to extend your simulation's capabilities far beyond what proprietary software can offer.

Low Barriers to Entry and No Licensing Fees

Python is free, open-source, and widely accessible. There are no expensive licences, no hidden fees, and no recurring payments. Whether you're an individual learner, a start-up, or an established business, Python lets you start innovating without financial constraints.

Transparent and Reproducible Models

Python's open-source nature means everything you create is fully transparent. You can show exactly how your model works, debug it with ease, and share it with others. This transparency ensures your simulations are robust, reproducible, and trusted by stakeholders.

Rapid Iteration and Innovation

With Python, iteration is fast. You can tweak parameters, test new ideas, and improve your models at a pace that proprietary tools simply can't match. This speed is invaluable in dynamic industries where agility is key.

Transferable Skills and Lasting Value

Python is one of the most popular programming languages in the world. By learning simulation in Python, you're not just acquiring a niche skill – you're building a foundation that can open doors across a range of fields, from data science to software development.

Python's advantages extend beyond technical benefits. It empowers engineers to take ownership of their work, freeing them from the constraints of traditional tools and unlocking a new level of creativity and efficiency. For me, Python wasn't just a tool – it was a gateway to greater professional freedom and success.

Why I Love Simulation Engineering

Simulation engineering has been transformative for me – not just as a career but as a way of working and living. Here's why I'm so passionate about it:

Flexibility and Remote Work

Simulation engineering isn't tied to a specific location or rigid schedule. Once I built my skills and reputation, I found the freedom to work remotely from anywhere in the world. Today, I enjoy a flexible lifestyle that allows me to balance meaningful work with spending time with my family and embracing life's adventures.

Interesting, Valued Work

No two projects are ever the same in simulation engineering. One day, I might be modelling transport systems; the next, I'm working on optimising mining operations or designing renewable energy systems. The variety keeps the work endlessly interesting, and the value I bring is always recognised. When you solve a problem that saves time, money, or resources, people notice.

Great Financial Rewards

Early in my career, I realised simulation engineers could earn more than just a good salary – they could earn life-changing money. By transitioning into contracting, I discovered how in-demand simulation skills were, and I learned how to command rates that allowed me to earn over £200k a year. This wasn't luck – it was about developing the right skills, building a reputation, and understanding my value in the market.

Simulation engineering is a rare field where you can combine technical creativity, problem-solving, and practical impact with a flexible and well-compensated career. For me, it's been the perfect fit, and I want to share this opportunity with others who are ready to embrace it.

Who This Book Is ForThis book is designed to meet you where you are and guide you to where you want to be, whether you're taking your first steps in simulation or aiming to level up your existing skills. If any of the following scenarios sound familiar, you're in the right place:

You're Looking to Add Simulation to Your Toolkit

You've heard of Python's potential for simulation and want to learn how to use it for your engineering projects. Whether it's optimising processes, forecasting outcomes, or building models to solve complex problems, this book will show you how.

You're at a Career Crossroads and Dreaming of Something New

Maybe you've reached a point where your current job feels stagnant, or you're considering a career that offers more flexibility, creativity, and financial freedom. Simulation engineering could be the fresh start you're looking for, and this book will guide you through the possibilities.

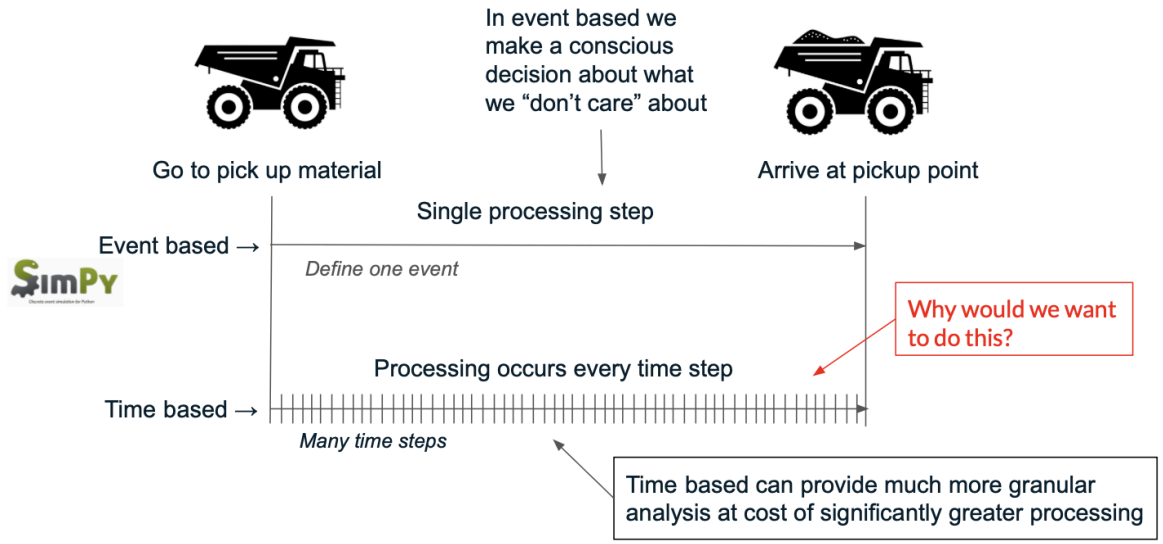
You're Already Using Simulation Software but Want to Make the Switch to Python

If you've worked with proprietary tools and felt the pain of high costs, limited flexibility, or opaque black-box models, this book will help you transition to Python. You'll learn how to replicate and improve what you've been doing – but without the constraints.

Simulation in Python is an exciting, empowering skill that can open doors to new opportunities and career paths. Whatever your motivation for learning, this book will give you the tools and knowledge you need to succeed.

What is Discrete-Event Simulation (DES)?

At its core, Discrete-Event Simulation is a method of modelling a system as a series of discrete events occurring at specific points in time. Each event triggers a change in the state of the system - like a machine starting or stopping, a customer arriving or leaving, or a signal being sent or received. By focusing on these critical junctures, DES provides a granular view of system dynamics without getting lost in the continuous flow of time.



School of Simulation

Key Characteristics of DES

- **Event-Driven Dynamics:** In DES, nothing happens between events. Changes occur only at specific points when events take place—like the ticking hands of a clock that move only when prompted.
- **Time Progression:** Time leaps from one event to the next, ignoring the silent intervals in between. This jump-forward mechanism ensures efficiency by concentrating computational efforts on moments of change.
- **Efficient Modelling:** By simulating only the events that alter the system's state, DES efficiently handles complex, real-world systems without unnecessary computational overhead.

Applications of DES

DES finds its strength in industries where timing and resource allocation are critical. Its applications are as diverse as the systems it models:

- **Manufacturing & Production:** Optimising production lines, scheduling maintenance, and managing inventory to reduce downtime and increase throughput.
- **Healthcare:** Modelling patient flow in hospitals to improve wait times, staff allocation, and resource utilisation.
- **Logistics:** Streamlining supply chain operations, from warehouse management to delivery routing, ensuring goods move efficiently from origin to destination.
- **Telecommunications:** Managing network traffic, predicting congestion points, and optimising data flow to enhance user experience.
- **Infrastructure Design:** Exploring "what-if" scenarios in large capital projects to make informed decisions early in the design lifecycle, saving time and resources.

Advantages of DES

- **Precision in Resource Allocation:** DES allows for meticulous simulation of systems with limited resources—ensuring every machine, worker, or component is utilised optimally.
- **Scenario and "What-If" Testing:** Easily test multiple scenarios to identify potential bottlenecks or areas for improvement, enabling proactive decision-making.
- **Cost-Effectiveness:** By modelling complex systems virtually, DES reduces the need for expensive real-world experiments, saving both time and money.

Limitations of DES

- **High Initial Effort:** Building a detailed simulation model requires significant upfront work to define every event, state, and interaction—much like crafting the blueprint before constructing a skyscraper.
- **Unsuitable for Continuous Processes:** DES excels with discrete events but isn't ideal for systems where continuous change is paramount, such as fluid dynamics or temperature variations.

SimPy's Role in DES

Enter **SimPy** - a powerful, process-based discrete-event simulation framework for Python. SimPy provides a clean and straightforward way to model DES systems, leveraging Python's simplicity and versatility. With SimPy, you can define events, manage resources, and simulate processes in a way that's both intuitive and efficient, making it an excellent tool for academia and industry alike.

SimPy Basics

Installation

Imagine possessing the power to predict the behaviour of a bustling shop or the seamless operation of a factory—all from the comfort of your desk. SimPy, a lightweight and intuitive Python library for discrete-event simulations, allows you to do just that. To begin your journey into the world of simulation, you need only a simple command:

```
pip install simpy
```

Once installed, you can begin writing simulations in Python.

Core Concepts

SimPy revolves around a few key concepts that help model the behaviour of real-world systems.

Generator Functions - What Are They?

In Python, generator functions allow you to write functions that can yield values one at a time, rather than returning everything at once. This is crucial for simulations because it enables you to pause and resume processes, simulating how things happen over time.

Think of a generator function as a pause button on a video. When you hit pause, the video doesn't start over it picks up right where you left off when you hit play again. Similarly, generator functions in SimPy allow you to model processes that unfold over time, but with the ability to pause and wait for certain events (like a customer waiting for service or a machine being repaired) before resuming.

Here's a simple example:

```
def countdown(n):
```

```
    while n > 0:
    yield n
    n -= 1
```

In this example, the countdown function counts down from a given number, pausing after each step. Each time you call the function, it resumes where it left off, making it ideal for modelling time-based processes.

```
for i in countdown(5):
    print(i)
```

Output:

5

4

3

2

1

How SimPy Uses Generator Functions

In SimPy, generator functions represent the processes in your simulation. These processes can wait for events to occur, such as a machine becoming available or a customer being served, and then continue running when the conditions are met. This is what makes SimPy so powerful - it models real-world concurrency without the complexity of managing multiple threads.

For example, in a bank simulation, you could model customers arriving at random intervals and waiting to be served. Each customer would be a generator function that “yields” when they arrive and waits until a teller is available to serve them. Once served, the function resumes and simulates the customer leaving the bank.

Let’s take a look at how this works in practice:

```
import simpy

def customer(env, name, service_time):
        print(f'{name} arrives at the bank at {env.now}')
        with bank_counter.request() as req:
        yield req
        print(f'{name} starts being served at {env.now}')
        yield env.timeout(service_time)
        print(f'{name} leaves the bank at {env.now}')

env = simpy.Environment()
bank_counter = simpy.Resource(env, capacity=1)

# Add customers to the environment
env.process(customer(env, 'Customer 1', 5))
env.process(customer(env, 'Customer 2', 3))

# Run the simulation
env.run(until=20)
```

Simulation output:

Customer 1 arrives at the bank at 0

Customer 2 arrives at the bank at 0

Customer 1 starts being served at 0

Customer 1 leaves the bank at 5

Customer 2 starts being served at 5

Customer 2 leaves the bank at 8

In this example:

- Each customer arrives at the bank, requests a counter, and waits if necessary.
- The yield keyword allows the customer to wait until a counter is free, simulating a real-world queue.
- The simulation runs for a specified time (20 units), and we can see how each customer's interaction with the system plays out.

Why Generators Make SimPy So Powerful

Generators allow SimPy to handle complex, time-dependent processes in a simple and elegant way. With generators, you can:

- Pause a process without freezing the whole simulation.
- Model multiple entities acting simultaneously, like customers in a queue or machines in a factory.
- Create realistic delays, such as travel time, waiting for service, or processing time.

This ability to model concurrency makes SimPy an excellent choice for simulating systems where many events happen at the same time, or where multiple resources are being shared among many entities.

The SimPy Environment

- The **Environment** is at the heart of any SimPy simulation. It manages the simulation clock and controls the scheduling and execution of events.
- It tracks simulation time, which can be accessed through `env.now`.
- Processes, resources, and events are created and managed through the environment.

Example:

```
import simpy
```

```
env = simpy.Environment()
```

SimPy Processes

- In SimPy, processes are represented using Python's **generator functions**. These processes yield events, which are scheduled by the environment.
- A **process** could represent anything from a customer arriving at a store to a machine processing items. Processes can be paused (using `yield`) and resumed later.

Example:

```
def process_example(env):
    print(f"Process starts at time {env.now}")
    yield env.timeout(5)
    print(f"Process resumes at time {env.now}")
```

SimPy Events

- **Events** represent specific points where something happens in the simulation, such as a machine finishing a task or a resource being requested.
- **Timeouts** are the most basic type of event in SimPy. You can use `env.timeout(t)` to simulate the passing of time. Other events include resource requests or process completions.

Example:

```
def machine(env):
    print(f"Machine starts at {env.now}")
    yield env.timeout(3)  # Machine works for 3 units of time
    print(f"Machine stops at {env.now}")
```

SimPy Resources

Resources in SimPy model shared assets that processes compete for, such as servers, machines, or workers. Resources are essential when modelling systems with limited availability.

- **Requesting Resources:** When a process needs access to a resource, it issues a request using `resource.request()`.
- **Capacity:** Resources can have limited capacity, representing the number of entities (e.g., machines, workers) that can serve processes simultaneously.

Using `with resource.request() as req`:

This is the simpler, more concise way to manage resource requests. The `with` statement automatically handles the request and release of the resource.

Example:

```
resource = simpy.Resource(env, capacity=1)
with resource.request() as req:
    yield req  # Wait until the resource is available
    yield env.timeout(5)  # Use the resource for 5 units of time
```

Using `resource.request()` and `resource.release()` Explicitly

In more complex simulations, you might need more control over the timing of resource requests and releases. This can be done by explicitly managing these events.

Example:

```
def process_with_explicit_request(env, resource):
    # Request the resource
    req = resource.request()
    yield req  # Wait until the resource is available
    print(f"Resource acquired at {env.now}")

    # Simulate using the resource
    yield env.timeout(5)
    print(f"Process using resource at {env.now}")

    # Release the resource manually
    resource.release(req)
    print(f"Resource released at {env.now}")

# Simulation setup
env = simpy.Environment()
resource = simpy.Resource(env, capacity=1)
env.process(process_with_explicit_request(env, resource))
env.run(until=10)
```

Explanation

- **Requesting the Resource:**
  - `req = resource.request()` creates a request event but doesn't yield it immediately. This allows you to control exactly when the process should wait for the resource.
  - `yield req` waits until the resource is available.
- **Using the Resource:**
  - After acquiring the resource, the process simulates using it for 5 units of time with `yield env.timeout(5)`.
- **Releasing the Resource:**
  - `resource.release(req)` releases the resource explicitly once the process is done using it.

When to Use Explicit Resource Management

- **Complex Simulations:** In more complex scenarios, where you may need to request multiple resources at different times, or where the resource usage depends on conditions evaluated during the process.
- **Conditional Releases:** If the release of the resource depends on specific conditions or additional logic, explicitly managing the resource gives you the flexibility to handle these scenarios.

Summary

- **`with resource.request() as req`:** Simplifies resource management by automatically handling request and release within the block.
- **Explicit `request()` and `release()`:** Provides more control over the timing and conditions of resource management, useful in complex simulations.

By understanding both approaches, you can choose the one that best fits the complexity and requirements of your simulation.

Simulation Flow

Process Creation: Processes are added to the environment using `env.process()` like so: `env.process(process_example(env))`

Simulation Execution: The environment runs until a given time or until there are no more events left using `env.run()`. This ensures that all events are processed in the correct order. The run time is specified like so:
`env.run(until=10)`

Note that time is unitless in simpy - it is up to you to decide what units of time you wish to use. Just remember to be consistent!

Summary of Key Functions

- `env.timeout(t)` — Simulates the passage of time (t time units).
- `env.process(func)` — Starts a process by adding it to the simulation.
- `env.run(until=t)` — Runs the simulation until time t or until there are no more events.
- `resource.request()` — Requests access to a resource.
- `resource.release()` — Releases a resource once a process is finished.

Writing a Simple SimPy Program

This section walks through the creation of a basic SimPy simulation, introducing key concepts like processes, timeouts, and events. Here's how to write a simple simulation using SimPy.

Step 1: Define the Environment

First, create a SimPy environment, which is responsible for managing simulation time and processing events. Every simulation begins by defining the environment:

```
import simpy

env = simpy.Environment()
```

Step 2: Define the Process

Processes in SimPy are represented by **Python generator functions**, where the `yield` keyword is used to pause the process until a specific event occurs (such as the passage of time).

Let's define a process where a car alternates between parking and driving:

```
def car(env):
    while True:
        print(f'Car parks at {env.now}')
        yield env.timeout(5)  # Car is parked for 5 units of time
        print(f'Car drives at {env.now}')
        yield env.timeout(2)  # Car drives for 2 units of time
```

Here:

- `env.timeout(5)` tells the simulation to pause the car process for 5 units of time (simulating that the car is parked).
- After 5 units, the process resumes and simulates the car driving for 2 units of time.

Step 3: Add the Process to the Environment

To add a process to the environment, you use `env.process()`. This schedules the process to start immediately:

```
env.process(car(env))
```

Step 4: Run the Simulation

To run the simulation, use `env.run()`. This runs the simulation for 15 units of time, during which the car will alternate between parking and driving.

```
env.run(until=15)
```

Complete Simple Example

```
import simpy

def car(env):
    while True:
        print(f'Car parks at {env.now}')
        yield env.timeout(5)  # Car is parked for 5 units of time
        print(f'Car drives at {env.now}')
        yield env.timeout(2)  # Car drives for 2 units of time

env = simpy.Environment()
env.process(car(env))
env.run(until=15)
```

Explanation of Output

The simulation will produce output like:

```
Car parks at 0
Car drives at 5
Car parks at 7
Car drives at 12
```

Here's what's happening:

1. The car parks at time `0` and stays parked for 5 time units.
2. It then drives from time `5` to time `7`.
3. This alternation between parking and driving continues until the simulation reaches time `15`.

What's Happening Under the Hood

- **Events:** The `timeout()` method creates events that the environment schedules. When a process yields an event, it pauses until that event is processed.
- **Process Control:** The `while True:` loop allows the car process to repeat indefinitely (or until the simulation stops). The process only resumes after each `yield` is completed.

Customising the Simulation

You can modify the simulation by:

1. Changing the time values in `env.timeout()`.
2. Adding more processes (e.g., more cars or different entities).
3. Introducing resources to simulate competition for limited assets (like parking spaces).

Key Components in SimPy

SimPy makes it simple to model real-world processes and systems by providing several fundamental components. Understanding these core elements will help you build more complex simulations.

Events and Processes

- **Events** are the building blocks of any SimPy simulation. An event represents something that happens at a specific point in time, such as a task being completed or a resource becoming available.
- **Processes** are special types of functions in SimPy that yield events. They simulate activities over time and can be paused and resumed using the `yield` keyword. Processes are typically represented as Python generator functions, which makes them ideal for simulations.

Example:

```
def machine(env):
    print(f'Machine starts at {env.now}')
    yield env.timeout(3)  # Machine operates for 3 time units
    print(f'Machine finishes at {env.now}')
```

Here, the machine operates for 3 time units, simulated by the `env.timeout()` function, which represents the passage of time.

Timeouts (Pausing Processes)

SimPy's **timeout** event allows a process to pause for a certain number of time units, simulating real-world delays. When you yield a timeout, the process pauses until the time elapses, then resumes.

Example:

```
def worker(env):
    print(f'Worker starts at {env.now}')
    yield env.timeout(2)  # Worker works for 2 time units
    print(f'Worker finishes at {env.now}')
```

In this example, the worker stops working after 2 units of time. The simulation time advances by 2 time units as the timeout is yielded.

Resources

Resources in SimPy model limited assets like machines, workers, or servers that processes compete for. When a process needs access to a resource, it issues a request, and when it's done, it releases the resource.

- **Requesting a resource:** You can request access to a resource using `resource.request()`. The process will wait until the resource becomes available.
- **Releasing a resource:** Once the process has finished using the resource, it releases it with `resource.release()`.

Example (Modelling a single worker handling tasks):

```
import simpy

def task(env, worker):
    with worker.request() as req:
        yield req  # Wait until the worker is available
        print(f'Task starts at {env.now}')
        yield env.timeout(3)  # Task takes 3 time units
        print(f'Task finishes at {env.now}')

env = simpy.Environment()
worker = simpy.Resource(env, capacity=1)  # Only one worker
available
env.process(task(env, worker))
env.run(until=10)
```

In this example:

- The task waits for the worker to become available.
- The worker is then used for 3 time units to complete the task.
- The capacity of `worker` is set to 1, meaning only one task can use the worker at any given time.

Scheduling Events

SimPy allows you to schedule future events and control how they're processed by the environment.

- **Scheduling with `timeout`:** The `env.timeout()` function is used to schedule an event for some time in the future.
- **Running until an event occurs:** You can run the simulation until a specific event using `env.run()`.

Example:

```
def event_scheduler(env):
    print(f'Starting at {env.now}')
    yield env.timeout(5)  # Schedule an event 5 units from now
    print(f'Event occurred at {env.now}')

env = simpy.Environment()
env.process(event_scheduler(env))
env.run(until=10)
```

In this example, an event is scheduled to occur after 5 units of time, and the environment is run until time `10`.

Combining Resources and Processes

You can model more complex scenarios by combining resources and processes. For example, multiple processes can compete for limited resources, representing real-world systems such as queues or production lines.

Example (Multiple processes sharing a single resource):

```
def task(env, worker, task_id):
    with worker.request() as req:
        yield req  # Wait until the worker is available
        print(f'Task {task_id} starts at {env.now}')
        yield env.timeout(2)  # Task takes 2 time units
        print(f'Task {task_id} finishes at {env.now}')

env = simpy.Environment()
worker = simpy.Resource(env, capacity=1)  # Only one worker
available
```

```
# Start multiple tasks
for i in range(3):
    env.process(task(env, worker, i))

env.run(until=10)
```

Here:

- Three tasks are created, but since there's only one worker, the tasks must wait for the worker to be available.
- Each task takes 2 time units, and the simulation runs for 10 units of time.

Simulating a Queue System

In many real-world systems, entities (like customers, jobs, or products) need to wait in line for a resource to become available (e.g., a service desk, a machine, or a server). SimPy makes it easy to simulate these kinds of queue systems using resources and processes.

The Basics of Queuing Systems

In a queuing system:

- **Entities** (e.g., customers or jobs) arrive at a service point.
- If the service is **busy**, the entity **waits in a queue**.
- Once the resource is **available**, the entity is served.
- After being served, the entity leaves, and the next entity in the queue is served.

Defining the Customer Process

Let's simulate a simple system where multiple customers arrive and compete for access to a single service desk (or "resource" in SimPy terms). Each customer will either be served immediately or will wait in a queue if the service desk is busy.

Customer Process Example:

```
import simpy

def customer(env, name, counter):
    print(f'{name} arrives at time {env.now}')
    with counter.request() as req:  # Request the service counter
        yield req  # Wait until the counter is available
        print(f'{name} is being served at time {env.now}')
        yield env.timeout(5)  # Simulate service time
        print(f'{name} leaves at time {env.now}')
```

In this example:

- The `customer` function models the behaviour of a customer:
    - The customer arrives at the service counter.
    - They wait for the resource (counter) to become available using `counter.request()`.
    - Once served, they spend `5` time units at the service desk.

- After being served, they leave the system.

Defining the Resource (Service Counter)

In SimPy, resources represent things that processes (like customers) compete for. Resources can have limited capacity, which makes them ideal for simulating service desks, machines, or workers.

```
counter = simpy.Resource(env, capacity=1)  # Only one service
counter available
```

Here:

- We define a resource `counter` with a capacity of `1`, meaning only one customer can be served at a time.
- If multiple customers request the counter at the same time, the rest will wait in a queue.

Adding Multiple Customers

Now, let's create multiple customers who will arrive at different times and request service from the counter.

Example:

```
env = simpy.Environment()
counter = simpy.Resource(env, capacity=1)

# Create 3 customers arriving at different times
env.process(customer(env, 'Customer 1', counter))
env.process(customer(env, 'Customer 2', counter))
env.process(customer(env, 'Customer 3', counter))

env.run(until=15)  # Run the simulation for 15 time units
```

Introducing Stochastic Delays Between Customer Arrivals

In real life, customer arrivals usually follow some form of stochastic process. A common way to model the time between arrivals is by using an exponential distribution. We can simulate this by introducing a stochastic delay between customer arrivals using `numpy` to generate random inter-arrival times, and `env.timeout()` in SimPy to handle the delays.

```
import simpy

import numpy as np


def customer(env, name, counter):

    print(f'{name} arrives at time {env.now}')

    with counter.request() as req:

        yield req

        print(f'{name} is being served at time {env.now}')

        yield env.timeout(5)  # Service time

        print(f'{name} leaves at time {env.now}')


def customer_generator(env, counter, mean_interarrival_time):

    for i in range(5):

        interarrival_time =
np.random.exponential(mean_interarrival_time)

        yield env.timeout(interarrival_time)  # Stochastic delay
between arrivals

        env.process(customer(env, f'Customer {i+1}', counter))


env = simpy.Environment()
```

School of Simulation

```
counter = simpy.Resource(env, capacity=1)


# Generate customers with stochastic delays between arrivals

mean_interarrival_time = 3  # Mean of the exponential distribution

env.process(customer_generator(env, counter,
mean_interarrival_time))


env.run(until=20)
```

Key Differences

- **Stochastic Inter-arrival Times:** The time between customer arrivals is now drawn from an exponential distribution using `np.random.exponential(mean_interarrival_time)`, where `mean_interarrival_time` is the average delay between arrivals.
- **More Realistic Customer Flow:** This models a more realistic arrival pattern, with customers arriving at irregular intervals, rather than fixed intervals of 3 time units as in the original example.

Understanding the Output

The output of the simulation will vary because of the random nature of the exponential distribution. However, it will generally follow the same pattern as before, with customers arriving, potentially waiting in a queue if the counter is busy, and leaving after their service time.

Example output might look like this:

```
Customer 1 arrives at time 2.1

Customer 1 is being served at time 2.1

Customer 2 arrives at time 4.3
```

```
Customer 1 leaves at time 7.1

Customer 2 is being served at time 7.1

Customer 3 arrives at time 8.6

Customer 2 leaves at time 12.1

Customer 3 is being served at time 12.1

Customer 4 arrives at time 13.2
```

Key Observations:

- **Variable Arrival Times:** Unlike in the fixed interval case, the times between customer arrivals are now random.
- **Queue Behaviour:** Similar to the fixed case, customers may arrive while another is being served and will queue up, waiting for their turn at the service counter.

Visualising the Queue

Visualising queue data is a powerful way to gain insights into the performance of a system, especially in larger simulations where the number of customers, jobs, or processes waiting in a queue can change significantly over time. By plotting this data, you can better understand the **load on resources**, identify **bottlenecks**, and optimise system efficiency.

Here's how visualising the queue can be beneficial, followed by an example of how to implement it in Python.

Why Visualise the Queue?

1. **Understand Resource Utilisation:**
    - Visualising the number of customers in the queue at different time intervals gives you a clear picture of how much strain is placed on your system resources. For example, if your queue frequently builds up, it might indicate that your service process is a bottleneck or that more resources are needed to handle the workload.
2. **Monitor Queue Fluctuations:**

- The queue size may fluctuate significantly based on arrival rates and service times. A visual representation helps track when queues grow and shrink, allowing you to identify peak load periods, times when customers are left waiting too long, or when resources are idle.

3. **System Performance Analysis:**
   - By visualising queues, you can analyse performance metrics like **average wait time**, **queue length over time**, and **peak queue size**. This allows you to fine-tune your system and optimise resource allocation.
4. **Improve Decision-Making:**
   - If your queue is consistently long, it could signal a need for more resources (e.g., additional servers or machines). Conversely, if the queue is often empty or short, it might indicate over-provisioning of resources. Visualisation aids in these operational decisions.

Example: Plotting the Number of Customers in the Queue Over Time

Let's expand our previous simulation to track the number of customers waiting in the queue over time. We'll use Python's **Matplotlib** to create a plot that shows how the queue size evolves during the simulation.

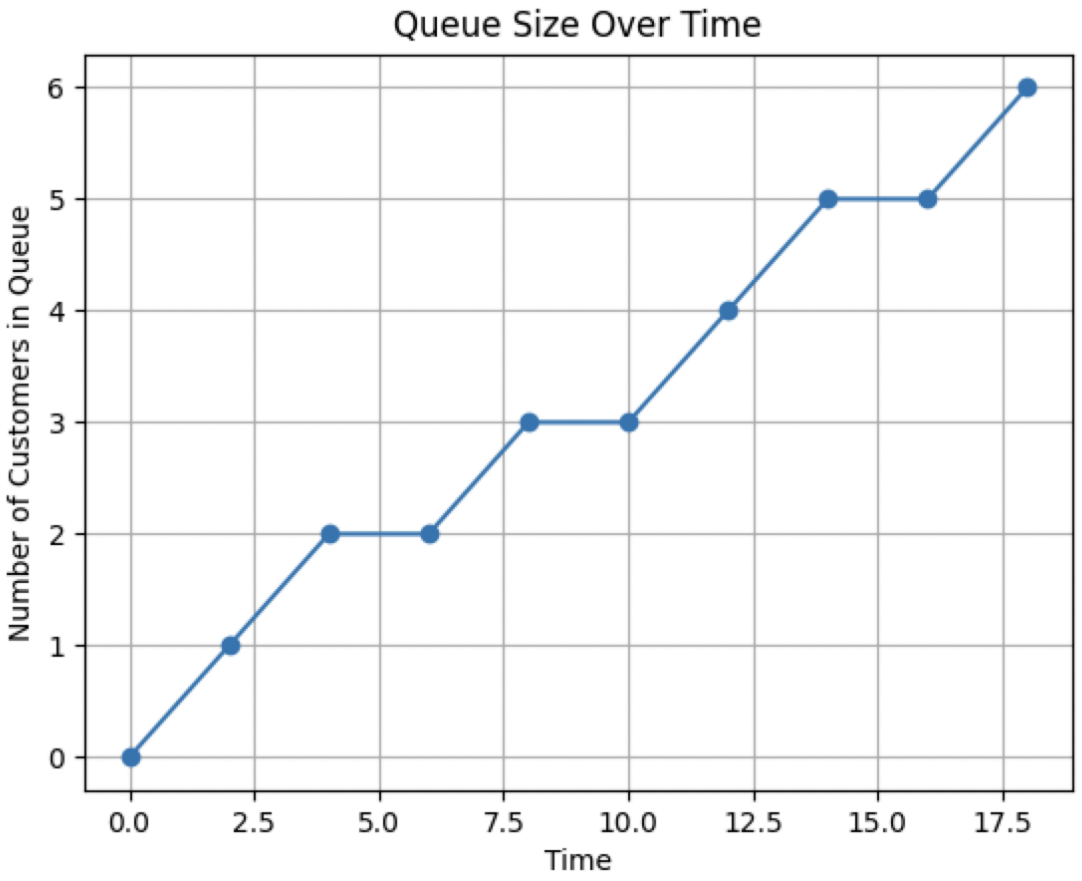
Code Implementation:

```
import simpy

import matplotlib.pyplot as plt



# Define the customer process

def customer(env, name, counter, wait_times, queue_lengths):

    arrival_time = env.now

    with counter.request() as req:

        queue_lengths.append((env.now, len(counter.queue)))  # Track
the queue length

        yield req
```

```
        wait_times.append(env.now - arrival_time)  # Track how long
the customer waited

        yield env.timeout(5)  # Service time


# Generate customers over time

def customer_generator(env, counter, wait_times, queue_lengths):

    for i in range(10):

        env.process(customer(env, f'Customer {i}', counter,
wait_times, queue_lengths))

        yield env.timeout(2)  # Customer arrival every 2 units of
time


# Create the simulation environment

env = simpy.Environment()

counter = simpy.Resource(env, capacity=1)  # Resource with a single
server


# Lists to track wait times and queue lengths

wait_times = []

queue_lengths = []


# Start the customer generation process

env.process(customer_generator(env, counter, wait_times,
queue_lengths))
```

School of Simulation

```
env.run(until=20)


# Extract the time points and queue lengths for plotting

times, queue_sizes = zip(*queue_lengths)


# Plot the number of customers in the queue over time

plt.plot(times, queue_sizes, marker='o')

plt.title('Queue Size Over Time')

plt.xlabel('Time')

plt.ylabel('Number of Customers in Queue')

plt.grid(True)

plt.show()
```



Explanation of the Code

- **Tracking Queue Lengths:**
  - The line `queue_lengths.append((env.now, len(counter.queue)))` logs the current time (`env.now`) and the number of customers in the queue at that moment (`len(counter.queue)`). This is done every time a customer arrives and requests the resource.
- **Plotting the Data:**
  - After the simulation runs, we extract the time points and queue sizes using `zip(*queue_lengths)` and plot them using **Matplotlib**.
  - The resulting plot will show how the queue size changes over time, with markers indicating the points when customers arrive and leave the queue.

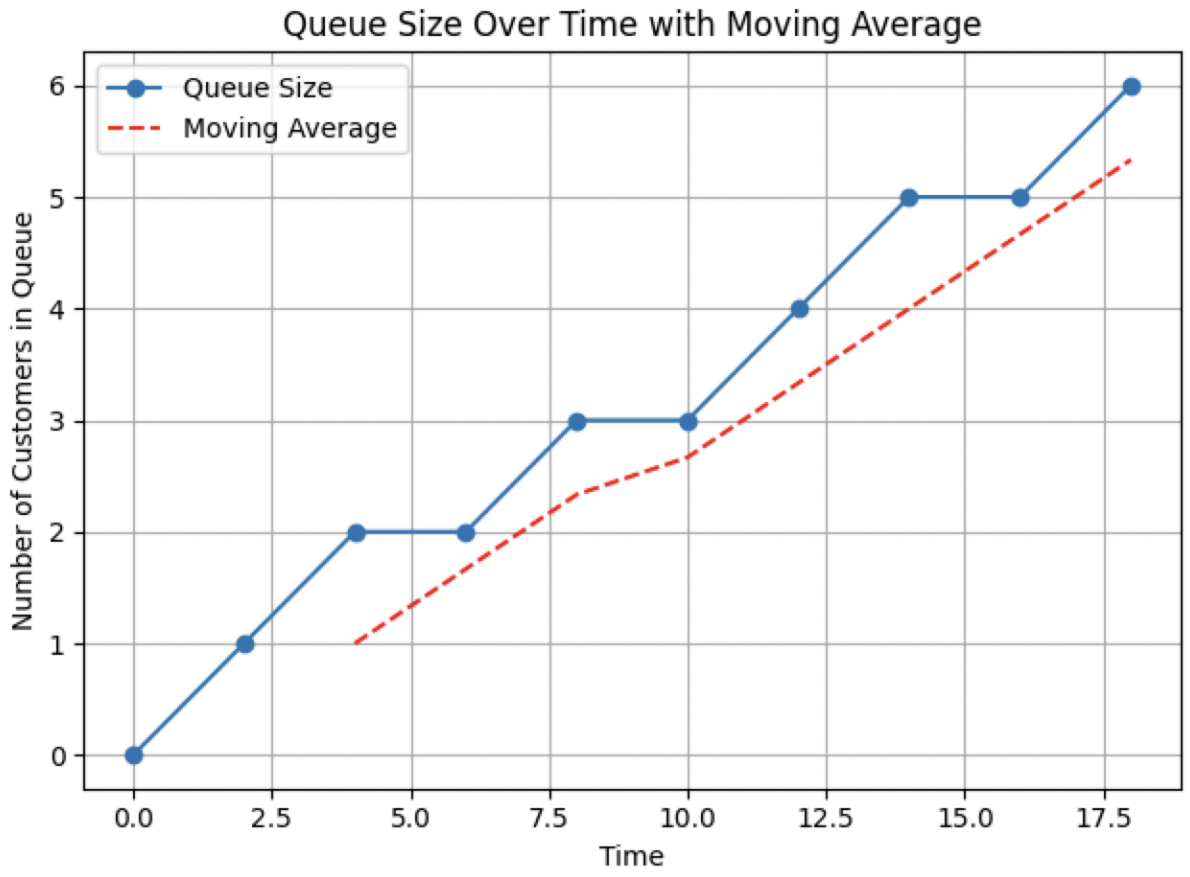
Customising the Plot

1. **Adding a Moving Average:**
    - You can smooth out the queue data by adding a moving average, which helps visualise trends over time rather than just individual points.

Example:

```
import numpy as np

# Plot the number of customers in the queue over time

plt.plot(times, queue_sizes, marker='o', label='Queue Size')



# Calculate and plot the moving average

window_size = 3

moving_avg = np.convolve(queue_sizes,
np.ones(window_size)/window_size, mode='valid')



# Adjust the x-axis times to align the moving average with the
correct starting point

adjusted_times = times[window_size-1:]  # Times for the moving
average plot

plt.plot(adjusted_times, moving_avg, label='Moving Average',
linestyle='--', color='red')



plt.title('Queue Size Over Time with Moving Average')

plt.xlabel('Time')

plt.ylabel('Number of Customers in Queue')

plt.legend()

plt.grid(True)
```

```
plt.tight_layout()

plt.show()
```

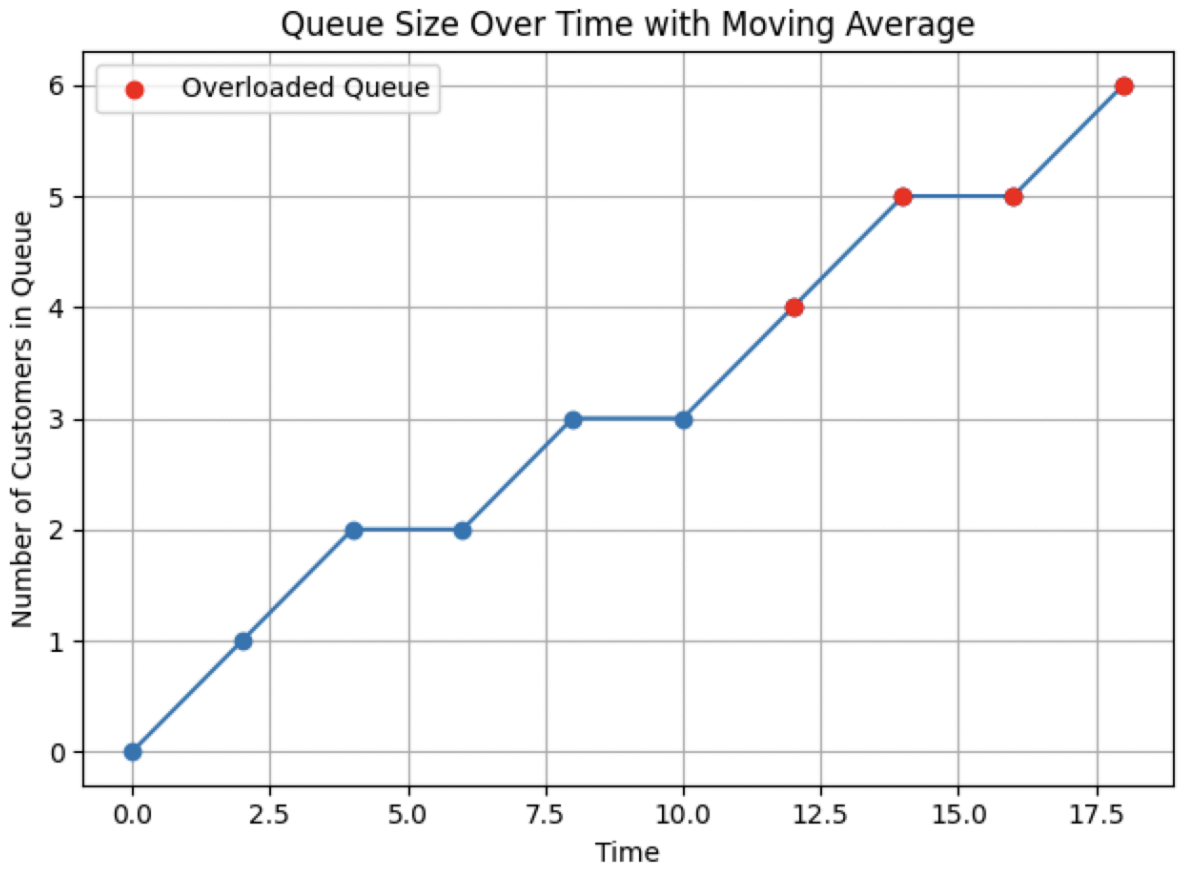


2. **Colour-Coding Queue Overload:**
    - Highlight periods where the queue exceeds a certain threshold (e.g., more than 3 customers in the queue) by adding different colours to the plot.

Example:

```
plt.plot(times, queue_sizes, marker='o')

overload_times = [t for t, q in zip(times, queue_sizes) if q > 3]

overload_sizes = [q for q in queue_sizes if q > 3]

plt.scatter(overload_times, overload_sizes, color='red',
label='Overloaded Queue', zorder=5)
```

School of Simulation

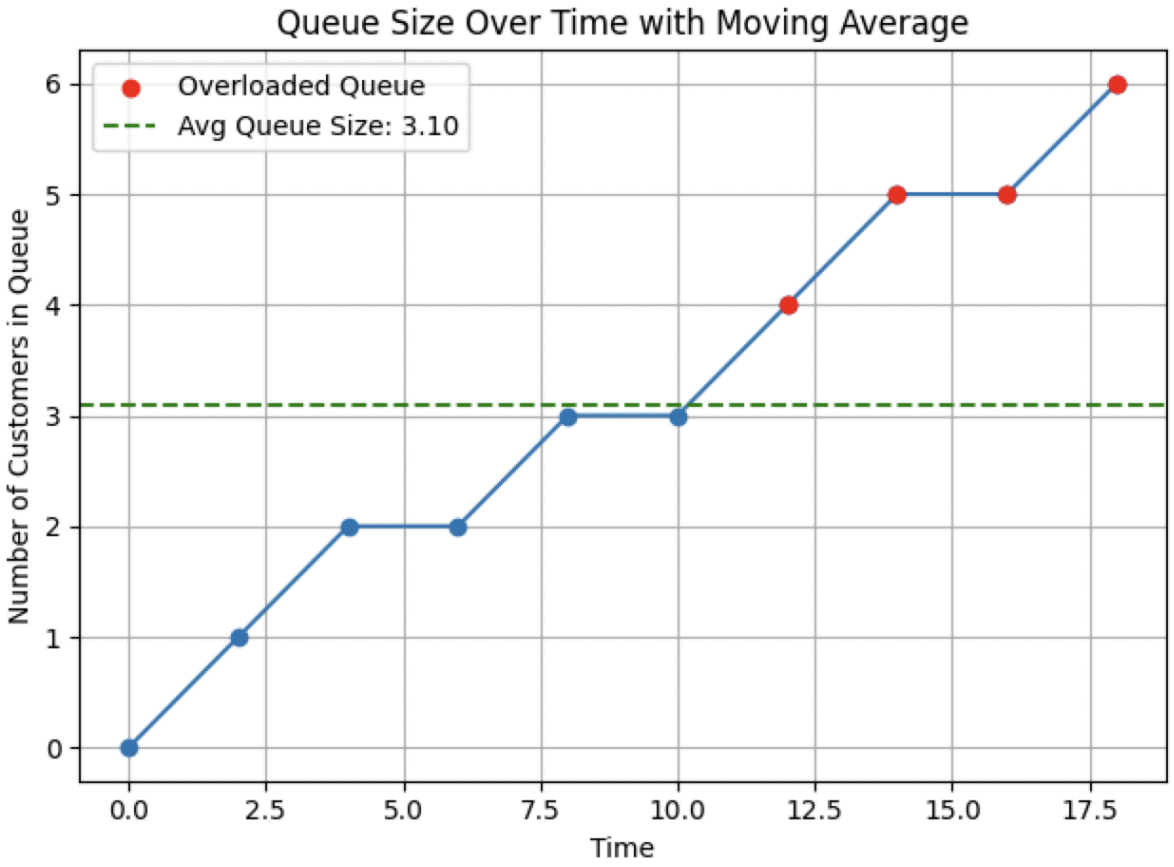


3. **Tracking Average Queue Size:**
    - You can calculate the average queue size and add a horizontal line to indicate it:

Example:

```
avg_queue_size = sum(queue_sizes) / len(queue_sizes)

plt.axhline(avg_queue_size, color='green', linestyle='--',
label=f'Avg Queue Size: {avg_queue_size:.2f}')
```

School of Simulation



Interpreting the Results

Once the plot is generated, you can begin to analyse:

- **Peaks in Queue Size:** High peaks in the graph indicate times when the system was overloaded, potentially causing long customer wait times.
- **Periods of Low Activity:** Flat or low points on the graph may indicate periods when the resource was idle or under-utilised.
- **Average Queue Size:** The overall trend of the queue size, along with the calculated average queue size, helps in determining whether the system's capacity is sufficient.

Practical Applications

- **Manufacturing Systems:** In manufacturing, queue visualisation can help identify bottlenecks in production lines, where items might be waiting too long to be processed.

- **Customer Service Systems:** For call centres or service desks, queue visualisation highlights periods of high traffic and customer wait times, allowing for better staffing decisions.
- **Logistics and Supply Chains:** In logistics, tracking queue sizes can help monitor shipping delays or warehouse processing times, improving overall efficiency.

Modelling Multiple Entities Competing for Resources with Traceability

In real-world systems, multiple entities often compete for the same resources, such as machines in a factory, workers in a service system, or servers in a network. By increasing the **capacity** of a resource (e.g., adding more workers or machines), we can manage the load more effectively. However, understanding how this affects **queue performance** requires careful **traceability**—tracking and visualising the behaviour of entities as they wait for and use resources.

This section demonstrates how to model multiple entities competing for a resource with **increased capacity** and how to **trace** the system's performance through both **print statements** and **visualisation**.

Adding Resources with Increased Capacity

Let’s modify our previous simulation to introduce a resource with more capacity (e.g., multiple servers or workers). We’ll track how customers interact with the resource and monitor the queue's size and resource utilisation over time.

Here’s an example where the resource (e.g., a service desk) has a capacity of 3, meaning up to three customers can be served simultaneously.

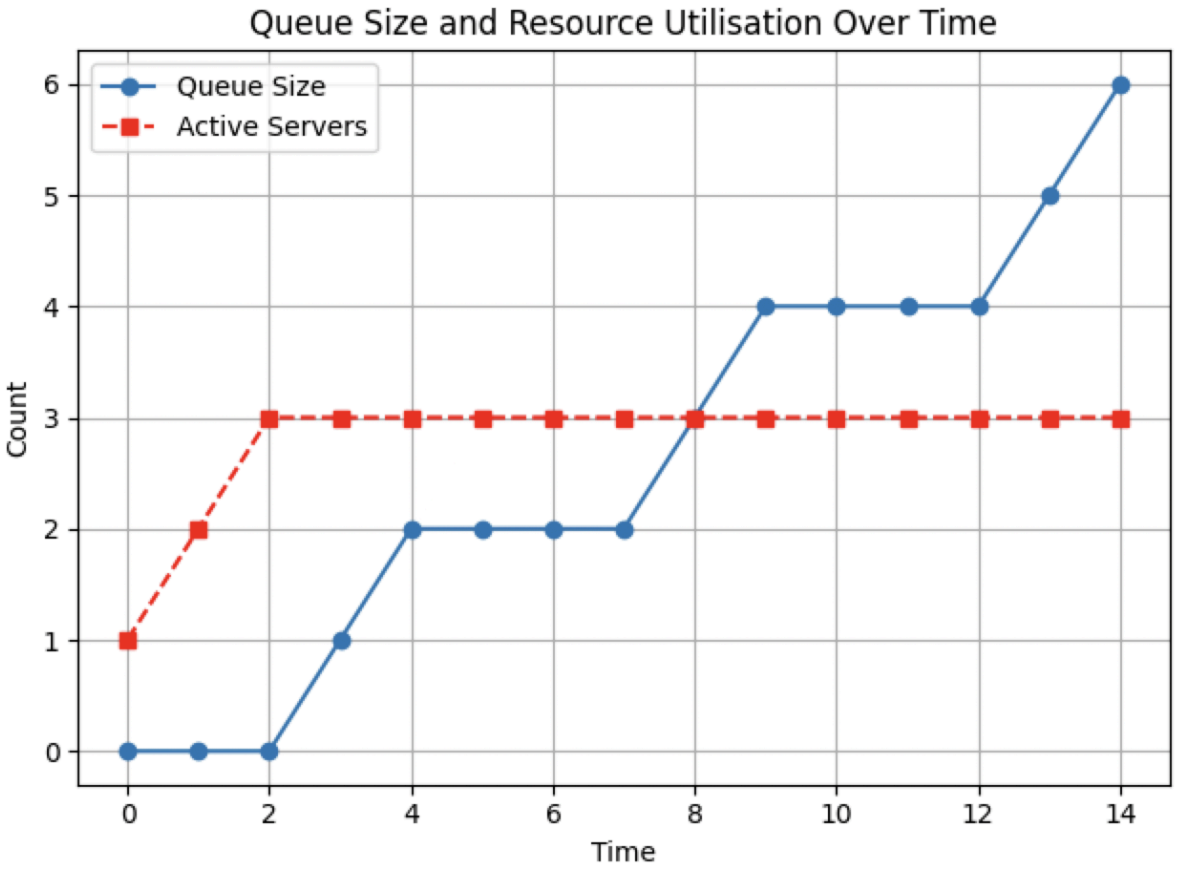
```
import simpy

import matplotlib.pyplot as plt



# Define the customer process

def customer(env, name, counter, wait_times, queue_lengths,
active_servers):
```

```
    arrival_time = env.now

    print(f'{name} arrives at time {env.now}')  # Traceability:
arrival time

    with counter.request() as req:

        queue_lengths.append((env.now, len(counter.queue)))  # Track
queue length

        active_servers.append((env.now, counter.count))  # Track
active servers

        yield req  # Wait for resource to become available

        wait_times.append(env.now - arrival_time)  # Track wait time

        print(f'{name} is being served at time {env.now}')  #
Traceability: start of service

        yield env.timeout(5)  # Simulate service time

        print(f'{name} leaves at time {env.now}')  # Traceability:
end of service


# Generate customers over time

def customer_generator(env, counter, wait_times, queue_lengths,
active_servers):

    for i in range(15):  # Simulate more customers

        env.process(customer(env, f'Customer {i}', counter,
wait_times, queue_lengths, active_servers))

        yield env.timeout(2)  # Customer arrival every 2 unit of
time
```

```
# Create the simulation environment

env = simpy.Environment()


# Define a resource with a capacity of 3 (e.g., 3 service desks)

counter = simpy.Resource(env, capacity=3)


# Lists to track wait times, queue lengths, and active servers

wait_times = []

queue_lengths = []

active_servers = []


# Start the customer generation process

env.process(customer_generator(env, counter, wait_times,
queue_lengths, active_servers))

env.run(until=30)


# Extract the time points and queue lengths for plotting

times, queue_sizes = zip(*queue_lengths)

times_servers, server_counts = zip(*active_servers)


# Plot the number of customers in the queue over time

plt.plot(times, queue_sizes, marker='o', label='Queue Size')
```

```
# Plot the number of active servers (resource utilisation) over time

plt.plot(times_servers, server_counts, marker='s', linestyle='--',
color='red', label='Active Servers')


plt.title('Queue Size and Resource Utilisation Over Time')

plt.xlabel('Time')

plt.ylabel('Count')

plt.legend()

plt.grid(True)

plt.tight_layout()

plt.show()
```

School of Simulation



Explanation of the Code

- **Resource with Capacity 3:**
  - `counter = simpy.Resource(env, capacity=3)` defines a resource (e.g., a service desk) with a capacity of 3, meaning up to 3 customers can be served simultaneously.
- **Traceability via Print Statements:**
  - **Arrival:** `print(f'{name} arrives at time {env.now}')` logs when each customer arrives.
  - **Start of Service:** `print(f'{name} is being served at time {env.now}')` logs when each customer begins service.
  - **End of Service:** `print(f'{name} leaves at time {env.now}')` logs when each customer leaves after being served.
- **Tracking Queue Lengths and Active Servers:**
  - **Queue Lengths:** `queue_lengths.append((env.now, len(counter.queue)))` tracks the number of customers waiting in the queue at each time point.

- **Active Servers:** `active_servers.append((env.now, counter.count))` tracks the number of servers (workers or machines) currently in use at each time point. This shows the resource utilisation over time.

Visualising the Results

After running the simulation, we can visualise two important aspects of the system's performance:

- **Queue Size Over Time:** This plot shows how many customers are waiting in the queue at different time points. Peaks in the plot indicate times when the system is overloaded.
- **Active Servers Over Time:** This plot shows the number of servers (or workers) actively serving customers over time. If the number of active servers frequently hits the resource capacity (e.g., 3), it indicates that the system is working at full capacity and might need more resources to avoid queue build-ups.

Analysing the System's Performance

By analysing the print statements and visualisation, you can get a clearer picture of how effectively the system is handling customer arrivals:

1. **Print Statement Traceability:**
   - The print statements provide a detailed trace of when each customer arrives, begins service, and leaves. This is useful for debugging and understanding the flow of events.
2. **Queue Visualisation:**
   - The queue size plot helps you see how the system handles customer arrivals. If the queue frequently builds up and customers are left waiting, it indicates that the system's resource capacity might be insufficient.
3. **Resource Utilisation Visualisation:**
   - The active servers plot shows how well the available resources (e.g., workers, machines) are utilised. If the plot frequently hits the capacity limit, you might need to consider increasing the capacity of your resources.

Practical Insights

By increasing the resource capacity and adding traceability, you can gain deeper insights into how multiple entities compete for resources in real-world systems. This is particularly useful for:

- **Manufacturing Systems:** Track how machines handle production loads and how often queues build up for resources.
- **Customer Service Systems:** Monitor how service agents are utilised and whether more agents are needed to prevent long customer wait times.
- **Network Systems:** Simulate how network servers handle multiple client requests and how server capacity impacts performance.

Advanced Resources

SimPy's Resource class is versatile, but it also provides more specialised resource types to model different real-world scenarios.

These include PriorityResource for prioritised access, PreemptiveResource for systems where higher-priority entities can interrupt lower-priority ones, and Container for resources that accumulate or deplete over time.

Priority Resources

In some systems, certain entities need to be prioritised over others. For example, in healthcare, critical patients are treated before those with less urgent conditions. SimPy's PriorityResource allows you to assign priorities to requests, ensuring that high-priority entities are served first, even if lower-priority entities arrived earlier.

Here's an example of how to use a PriorityResource in SimPy:

```
# Create a priority resource with 1 server
priority_teller = simpy.PriorityResource(env, capacity=1)

# Customer with a priority level
def priority_customer(env, name, priority, teller):
        print(f'{name} arrives at the bank at {env.now}')
        with teller.request(priority=priority) as req:
        yield req
        print(f'{name} starts being served at {env.now}')
        yield env.timeout(5)
        print(f'{name} leaves the bank at {env.now}')
```

In this example:

- The priority_customer function works just like the regular customer function, but it includes a priority argument. This priority determines the order in which customers are served.
- Lower numbers represent higher priority. So, a customer with priority 0 will be served before a customer with priority 1, even if the latter arrived earlier.

Preemptive Resources

Sometimes, high-priority entities need to interrupt ongoing activities. For example, in emergency services, an ambulance responding to a life-threatening call might preempt a vehicle that's already en route to a less critical case. SimPy models this scenario with PreemptiveResource.

With a PreemptiveResource, entities with higher priority can preempt (interrupt) lower-priority entities that are currently using the resource. The lower-priority entity can resume its activity later, once the higher-priority task is finished.

Here's an example:

```
# Create a preemptive resource with 1 server
preemptive_teller = simpy.PreemptiveResource(env, capacity=1)

# Customer with a priority level
def preemptive_customer(env, name, priority, teller):
        print(f'{name} arrives at the bank at {env.now}')
        with teller.request(priority=priority, preempt=True) as req:
        try:
        yield req
        print(f'{name} starts being served at {env.now}')
        yield env.timeout(5)
        print(f'{name} leaves the bank at {env.now}')
        except simpy.Interrupt as interrupt:
        print(f'{name} was interrupted at {env.now} by {interrupt.cause}')
```

In this example, higher-priority customers can interrupt lower-priority customers, ensuring that the most urgent needs are always met first.

Containers

For systems where resources accumulate or deplete over time, SimPy provides the Container class. Containers are used to model resources like fuel, inventory, or water levels, which can be added to or removed from over the course of the simulation.

Here's a basic example of a fuel tank model using a Container:

```
# Create a container with an initial level of 100 and a maximum capacity of 200
fuel_tank = simpy.Container(env, init=100, capacity=200)

# A process that consumes fuel
```

```
def consume_fuel(env, amount, fuel_tank):
        while True:
        yield fuel_tank.get(amount)
        print(f'Consumed {amount} fuel at {env.now}. Fuel level: {fuel_tank.level}')
        yield env.timeout(10)  # Wait 10 units of time before consuming more fuel
```

In this example:

- We define a fuel_tank container with an initial level of 100 units of fuel and a maximum capacity of 200 units.
- The consume_fuel process simulates fuel consumption. Each time the process runs, it requests a certain amount of fuel from the tank. If enough fuel is available, the process proceeds; otherwise, it waits until the tank is refilled.

Refilling the Container

Containers can also be refilled over time. Let's add a refilling process to our fuel tank example:

```
# A process that refills the fuel tank
def refill_fuel(env, amount, fuel_tank):
        while True:
        yield env.timeout(20)  # Wait 20 units of time between refills
        fuel_tank.put(amount)
        print(f'Refilled {amount} fuel at {env.now}. Fuel level: {fuel_tank.level}')
```

In this scenario:

- The refill_fuel process adds fuel to the tank every 20 units of time, ensuring that the fuel level doesn't drop too low.
- The combination of consumption and refilling allows us to model a system where resources fluctuate over time, providing a more realistic view of how the system operates.

Best Practices and Common Pitfalls

When working with resources in SimPy, there are a few best practices to keep in mind to ensure your simulations run efficiently and produce accurate results:

Monitor Resource Utilisation

Always track how your resources are being used. SimPy makes it easy to monitor resource utilisation, which helps you identify bottlenecks in your system. For example, if you see that

a resource is being used 100% of the time, it might indicate that the system is under-resourced, and adding more capacity could improve performance. On the other hand, if a resource is underutilised, it may be a sign that the system could be streamlined to reduce costs. You can monitor resource utilisation by tracking how often resources are requested and released, and how long entities wait for access to them. This data can be collected and analysed during or after the simulation to help inform decision-making.

Avoid Over-Complicating Resource Management

While it's tempting to model every possible detail, keep in mind that adding unnecessary complexity can slow down your simulation and make it harder to interpret the results. Focus on the critical resources that have the biggest impact on the system's performance. For instance, in a hospital simulation, you may not need to model every individual piece of equipment; instead, focus on major bottlenecks like operating rooms or critical care units.

Balance Resource Capacity

In real-world systems, resources often need to be balanced against demand. For example, adding more machines in a factory may reduce queue times, but if the factory lacks the workforce to operate them, the added capacity won't improve performance. Similarly, in a bank simulation, adding more tellers might reduce customer wait times, but it also increases operational costs. Use simulations to find the optimal balance between resource capacity and demand to ensure the system is both efficient and cost-effective.

Test Different Scenarios

One of the major benefits of simulation is the ability to test different scenarios without impacting the real world. In SimPy, it's easy to adjust parameters - such as resource capacities, process durations, or arrival rates - to see how they affect the system. By testing various configurations, you can find the best solution for your specific needs. For example, in a logistics network, you might test how different shipping schedules impact delivery times and costs, or in a manufacturing plant, you could experiment with different machine setups to optimise throughput.

Be Mindful of Deadlocks

Deadlocks occur when two or more processes are waiting for each other to release resources, and neither can proceed. For example, in a simulation of a manufacturing plant, two machines might each need the same tool to complete their tasks. If both machines request the tool at the same time and neither releases it, the simulation can get stuck. Prevent deadlocks by ensuring that processes request and release resources in a logical order and by designing your system so that it can handle potential resource conflicts gracefully.

Use Priority and Preemptive Resources When Necessary

As seen with PriorityResource and PreemptiveResource, there are times when some entities must be prioritised over others. In systems like emergency services or healthcare, where critical cases take precedence over less urgent ones, these advanced resource types can be invaluable. Use them strategically to ensure that high-priority tasks are completed first without sacrificing overall system performance.

Resource management is a key aspect of creating accurate and insightful simulations with SimPy. By modelling how entities compete for limited resources, you can gain a deeper understanding of system bottlenecks, resource utilisation, and the trade-offs involved in balancing capacity with demand.

Tips for Efficient Simulations

When working with simulations, especially for complex systems with multiple entities and events, it's essential to write efficient and manageable code. Here are some best practices to ensure your simulations run smoothly and remain easy to understand.

Write Modular Code

- **Break Down Your Simulation:** Split your simulation into smaller, more manageable components. Each process (like a machine, customer, or worker) should have its own function, and each function should be responsible for one specific task.

Example: Instead of writing everything in one process, divide your logic:

```
def serve_customer(env, counter):
    with counter.request() as req:
        yield req
        yield env.timeout(5)  # Service time

def customer_generator(env, counter):
    for i in range(5):
        yield env.timeout(3)
        env.process(serve_customer(env, counter))
```

This keeps each process focused, which makes debugging and extending the code easier.

Use Meaningful Variable Names

- **Clarity is Key:** Use descriptive variable names that indicate their role in the simulation. For example, name your resources based on their function (`worker`, `machine`, `counter`) and processes by what they do (`customer`, `task_generator`).

Bad Example:

```
def p(env, r):
    ...
```

Good Example:

School of Simulation

```
def customer(env, counter):
    ...
```

Manage Time Properly

- **Simulation Time vs. Real Time:** Remember that SimPy simulates time — it doesn't represent real-time. Adjust your time units (minutes, hours, etc.) based on the context of the simulation. Be clear about whether your time units represent seconds, minutes, or another measurement to avoid confusion.
- **Use `env.now`:** The `env.now` attribute keeps track of the current simulation time. Use this to log key events and better understand the progression of time in your simulation.

Control Simulation Length

- **Set Clear End Conditions:** Ensure that your simulation stops at an appropriate time. You can specify the stopping condition with `env.run(until=...)`. Simulations that aren't run long enough may not generate statistically significant results, and simulations that run too long may generate too much data or take too long to run.

Example:

```
env.run(until=100)  # Stops the simulation after 100 time units
```

Avoid Overloading Resources

- **Simulating Resource Availability:** Ensure that your resource capacities reflect real-world constraints. For example, if a worker can only handle one customer at a time, set `capacity=1`. Larger capacity values (e.g., `capacity=10`) simulate systems that can handle multiple entities simultaneously.

Example:

```
counter = simpy.Resource(env, capacity=1)  # Only one customer can
be served at a time
```

Track Statistics and Performance

- **Measure System Performance:** Use counters, logs, or tracking variables to collect statistics during the simulation. You can track things like wait times, service times, or queue lengths to assess the performance of your system.

Example:

```
wait_times = []

def customer(env, counter, wait_times):
    arrival_time = env.now
    with counter.request() as req:
        yield req
        wait_times.append(env.now - arrival_time)  # Track wait time
        yield env.timeout(5)
```

At the end of the simulation, you can analyse this data to find bottlenecks or optimise resource usage.

Visualise Your Simulation

- **Plot Your Results:** Visualising simulation results using libraries like `matplotlib` can help you better understand the behaviour of your system. For example, you can plot queue lengths, wait times, or the number of resources in use over time.

Example (Plotting wait times):

```
import matplotlib.pyplot as plt

plt.hist(wait_times, bins=10)
plt.title('Customer Wait Times')
plt.xlabel('Time Units')
plt.ylabel('Number of Customers')
plt.show()
```

Debugging and Validation

- **Print Statements:** Use print statements to log key events, especially during debugging. Logging the time (`env.now`) and important transitions (like when a customer arrives or leaves) helps you verify that your simulation behaves as expected.

Example:

```
print(f'Customer arrives at {env.now}')
```

- **Stepping Through Simulation:** SimPy provides methods like `env.step()` to run the simulation one event at a time. This can be useful for debugging complex interactions.

Example:

```
env.step()  # Executes one event at a time
```

Plan for Scalability

- **Efficiency for Large Simulations:** As the complexity of your simulation increases, with more entities and events, performance can degrade. To avoid performance bottlenecks:
    - Use efficient data structures (e.g., lists, queues).
    - Limit the scope of your simulation (e.g., simulate only key parts of a system).
    - Avoid excessive logging or printing in large simulations, as it can slow down execution.

Analysing and Visualising Simulation Data

A critical aspect of building simulations is analysing the data that comes out of them. It's not enough to simply run a simulation - you need to understand the insights it provides.

The data generated by your simulation can reveal bottlenecks, inefficiencies, and opportunities for improvement. In this chapter, we'll explore how to collect, analyse, and interpret simulation data to draw actionable conclusions.

Collecting Simulation Data

Before you can analyse data from a simulation, you need to collect it effectively. SimPy provides straightforward methods to capture data as the simulation runs, allowing you to store and later evaluate metrics like wait times, resource utilisation, or throughput.

What Data Should You Collect?

The specific data you collect depends on the problem you're trying to solve. In most cases, you'll want to focus on key performance indicators (KPIs) that reflect the efficiency and effectiveness of the system. Common metrics include:

- Wait times: How long do entities (e.g. customers, patients, tasks) wait before receiving service or processing?
- Queue lengths: How many entities are waiting in line at any given moment?
- Resource utilisation: How often are resources (e.g. machines, staff, vehicles) in use, and how much idle time do they have?
- Throughput: How many entities are processed or served over a given period?

Example: Data Collection in a Factory Simulation

Let's say you're modelling a simple factory that processes products through two machines. Each product must pass through Machine 1 before moving on to Machine 2. The goal is to understand how long each product spends in the system and how efficiently the machines are used.

Here's how to set up data collection for this factory simulation:

```
import simpy
import random
```

```
# Data collection lists
product_wait_times = []
machine_utilisation = {'Machine 1': 0, 'Machine 2': 0}

# Product process
def product(env, name, machine1, machine2):
        arrival_time = env.now
        print(f'{name} enters the factory at {env.now}')

        # Request and use Machine 1
        with machine1.request() as req1:
        yield req1
        start_time_machine1 = env.now
        yield env.timeout(random.uniform(5, 10))  # Process time on Machine 1
        machine_utilisation['Machine 1'] += env.now - start_time_machine1

        # Request and use Machine 2
        with machine2.request() as req2:
        yield req2
        start_time_machine2 = env.now
        yield env.timeout(random.uniform(3, 7))  # Process time on Machine 2
        machine_utilisation['Machine 2'] += env.now - start_time_machine2

        # Record the total time spent in the system
        total_time = env.now - arrival_time
        product_wait_times.append(total_time)
        print(f'{name} leaves the factory at {env.now}, total time: {total_time}')

# Set up the simulation environment
env = simpy.Environment()

# Create resources for the machines
machine1 = simpy.Resource(env, capacity=1)
machine2 = simpy.Resource(env, capacity=1)

# Add products to the simulation
for i in range(5):
        env.process(product(env, f'Product {i+1}', machine1, machine2))

# Run the simulation for 50 units of time
env.run(until=50)
```

Example Output

```
Product 1 enters the factory at 0
Product 2 enters the factory at 0
Product 3 enters the factory at 0
Product 4 enters the factory at 0
Product 5 enters the factory at 0
Product 1 leaves the factory at 9.462990500968086, total time: 9.462990500968086
Product 2 leaves the factory at 19.94591558290178, total time: 19.94591558290178
Product 3 leaves the factory at 29.334574750299282, total time: 29.334574750299282
Product 4 leaves the factory at 34.59649286985815, total time: 34.59649286985815
Product 5 leaves the factory at 42.10036910976831, total time: 42.10036910976831
```

Breakdown of the Example

- Data Collection: We collect two types of data - product wait times and machine utilisation. Product wait times are stored in the product_wait_times list, while the utilisation times for the machines are recorded in the machine_utilisation dictionary.
- Process Timing: Each product goes through two machines, with random processing times on each. The total time spent on each machine is recorded and added to the respective machine's utilisation time.
- Simulation Output: When the simulation finishes, we'll have a list of total times each product spent in the factory and data on how long each machine was in use.

Analysis

Once you've collected data from the simulation, the next step is to analyse it. Depending on the complexity of the simulation and the data collected, this can range from simple summary statistics to more advanced data visualisation and statistical analysis.

The simplest way to start analysing simulation data is by calculating summary statistics such as averages, medians, or standard deviations. These provide a quick overview of the system's performance.

For example, let's calculate the average wait time for products in the factory simulation:

```
# Calculate the average wait time
average_wait_time = sum(product_wait_times) / len(product_wait_times)
print(f'Average product wait time: {average_wait_time:.2f}')
```

Output:
Average product wait time: 24.39

Similarly, you can calculate the utilisation rate for each machine by dividing the total time the machine was in use by the total simulation time:

```
# Calculate utilisation rates
simulation_time = 50
utilisation_machine1 = machine_utilisation['Machine 1'] / simulation_time
utilisation_machine2 = machine_utilisation['Machine 2'] / simulation_time

print(f'Utilisation of Machine 1: {utilisation_machine1:.2%}')
print(f'Utilisation of Machine 2: {utilisation_machine2:.2%}')
```

Output:
Utilisation of Machine 1: 65.78%
Utilisation of Machine 2: 59.24%

In this example:

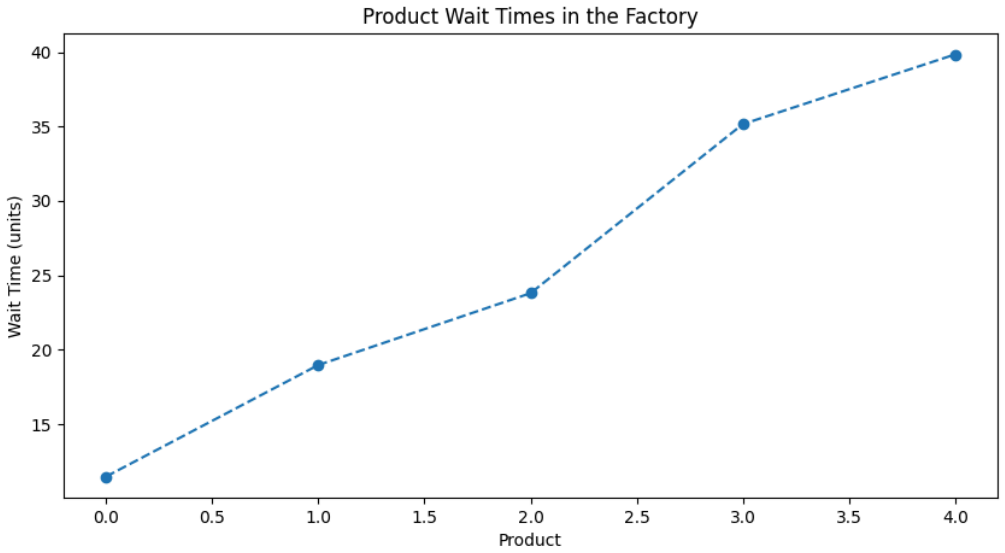
- Average Wait Time: We calculate the average time products spent in the factory, giving us a measure of overall system performance.
- Machine Utilisation: By calculating the percentage of time each machine was in use, we can identify whether the machines are underutilised or overworked.

Visualising Simulation Data

While summary statistics are helpful, visualising the data can provide deeper insights. Plotting wait times, queue lengths, or resource utilisation over time can help you identify trends, bottlenecks, or periods of high demand.

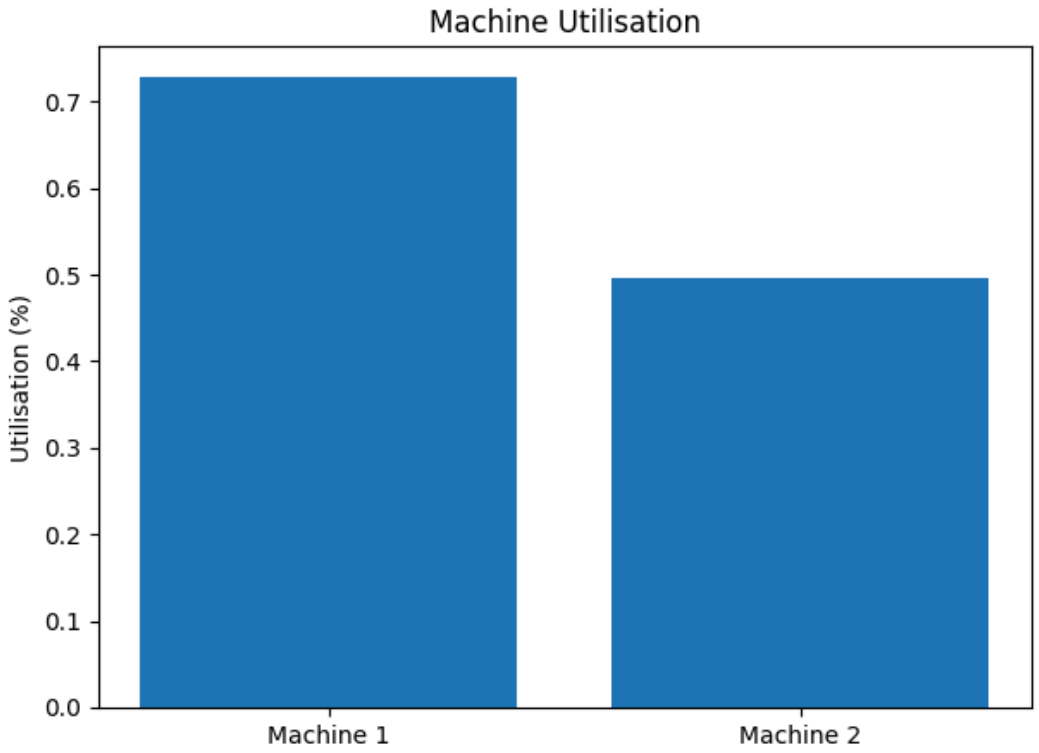
Let's use the matplotlib library to create some basic visualisations:

```
import matplotlib.pyplot as plt

# Plot product wait times
plt.figure(figsize=(10, 5))
plt.plot(product_wait_times, marker='o', linestyle='--')
plt.title('Product Wait Times in the Factory')
plt.xlabel('Product')
plt.ylabel('Wait Time (units)')
plt.show()
```




```
# Plot machine utilisation as a bar chart
machines = ['Machine 1', 'Machine 2']
utilisations = [utilisation_machine1, utilisation_machine2]

plt.figure(figsize=(7, 5))
plt.bar(machines, utilisations)
plt.title('Machine Utilisation')
plt.ylabel('Utilisation (%)')
plt.show()
```

School of Simulation



Interpreting the Results

- Product Wait Times: The line plot of product wait times shows how long each product spent in the factory. You might notice patterns - such as increasing wait times for later products - that could indicate bottlenecks in the system.
- Machine Utilisation: The bar chart of machine utilisation reveals how efficiently each machine was used. If one machine is underutilised compared to another, you might investigate why and whether changes in capacity or scheduling could improve overall efficiency.

Instrumenting Simulations and Logging

For more complex simulations, it's often useful to add logging or instrumentation to track specific events or metrics as the simulation runs. This can help you diagnose issues, understand the behaviour of the system, and make data-driven decisions about how to optimise it.

Using Custom Logging

In Python, you can use the built-in logging module to create detailed logs of your simulation's execution. This can be particularly useful for debugging or understanding how different processes interact with each other.

Here's an example of how to add custom logging to the factory simulation:

```
import logging

# Set up logging
logging.basicConfig(level=logging.INFO)

# Product process with logging
def product(env, name, machine1, machine2):
        arrival_time = env.now
        logging.info(f'{name} enters the factory at {env.now}')

        # Request and use Machine 1
        with machine1.request() as req1:
        yield req1
        start_time_machine1 = env.now
        yield env.timeout(random.uniform(5, 10))
        machine_utilisation['Machine 1'] += env.now - start_time_machine1
        logging.info(f'{name} finished at Machine 1 at {env.now}')

        # Request and use Machine 2
        with machine2.request() as req2:
        yield req2
        start_time_machine2 = env.now
        yield env.timeout(random.uniform(3, 7))
        machine_utilisation['Machine 2'] += env.now - start_time_machine2
        logging.info(f'{name} finished at Machine 2 at {env.now}')

        total_time = env.now - arrival_time
        product_wait_times.append(total_time)
        logging.info(f'{name} leaves the factory at {env.now}, total time: {total_time}')
```

With logging in place, you can track when each product enters and exits the factory, as well as when it finishes processing on each machine. This gives you a detailed view of how the simulation unfolds over time, which can be invaluable for debugging or understanding complex interactions between entities and resources.

Here's an example of what the log output might look like:

```
INFO:root:Product 1 enters the factory at 0
INFO:root:Product 1 finished at Machine 1 at 7.3
INFO:root:Product 1 finished at Machine 2 at 14.5
INFO:root:Product 1 leaves the factory at 14.5, total time: 14.5
INFO:root:Product 2 enters the factory at 2
INFO:root:Product 2 finished at Machine 1 at 10.5
INFO:root:Product 2 finished at Machine 2 at 17.8
INFO:root:Product 2 leaves the factory at 17.8, total time: 15.8
```

This log provides a step-by-step account of what happens to each product throughout the simulation. You can adjust the logging level to show more or less detail, depending on your needs.

Using Simulation Data for Decision Support

Once you've collected and analysed the data from your simulation, the next step is using that data to inform decisions. This could involve:

- System Optimisation: By identifying bottlenecks or underused resources, you can optimise your system's performance. For instance, if one machine has a much higher utilisation rate than others, you might consider adding more capacity to that machine or redistributing tasks to balance the workload.
- Scenario Testing: Simulations allow you to test different scenarios without real-world consequences. By adjusting parameters like resource capacity, process times, or arrival rates, you can evaluate how changes to the system would impact performance. This helps you make informed decisions about potential improvements or future expansions.
- Predictive Modelling: In industries like manufacturing, logistics, or healthcare, simulation data can be used to predict future performance. By analysing historical data and running simulations with varying assumptions, you can forecast potential challenges and prepare for them proactively.

Identifying Bottlenecks

A key outcome of simulation analysis is identifying bottlenecks—areas where entities are delayed or resources are overburdened. Bottlenecks can limit system throughput, increase wait times, and reduce overall efficiency. By understanding where these bottlenecks occur, you can implement targeted improvements that address the root causes of inefficiencies.

In the factory simulation example, bottlenecks might occur if one machine is used far more than the other, leading to long queues at that machine. By analysing machine utilisation and wait times, you can pinpoint the exact location of the bottleneck and explore solutions such as increasing machine capacity, reducing processing times, or balancing the workload more evenly.

How to Resolve Bottlenecks

Once you've identified a bottleneck, you can experiment with different strategies to resolve it. Here are some common approaches:

- Increase Capacity: If a resource (e.g. a machine, a doctor, a server) is over-utilised, increasing its capacity can reduce wait times and improve system throughput. In the factory example, adding another machine could reduce the queue length and speed up processing.
- Redistribute Workload: If one resource is overburdened while others are underutilised, redistributing tasks more evenly can improve efficiency. For instance, in a hospital simulation, if one department has a long queue while others have spare capacity, reallocating patients to the less busy departments can reduce overall wait times.
- Improve Process Efficiency: Reducing the time required for certain processes can help alleviate bottlenecks. For example, streamlining the steps in a manufacturing process or automating certain tasks can reduce the time entities spend waiting for resources, leading to improved system performance.

Advanced Data Collection Techniques

Effective data collection is crucial to gaining meaningful insights from your simulations. In this section, we'll explore advanced techniques for collecting, organising, and storing simulation data. Whether you're managing complex, hierarchical datasets or tracing specific events within a simulation, understanding how to collect and store data properly will ensure that you can analyse it effectively later.

Custom Data Structures

When simulations become more complex, organising the data generated by various entities and processes can become challenging. Simple lists or dictionaries may not suffice, especially when dealing with hierarchical or multi-dimensional data, such as tracking the progress of multiple entities across several stages. In these cases, custom classes or data structures provide a more flexible and scalable solution.

Organising Hierarchical Data

Consider a scenario where you are simulating a production line. Each product passes through multiple stages - assembly, testing, packaging - and each stage generates its own set of performance metrics like processing time, wait time, and resource usage. Using a custom class can help you capture this hierarchical data efficiently.

```
class Product:
        def __init__(self, product_id):
        self.product_id = product_id
        self.stages = {}

        def add_stage_data(self, stage_name, processing_time, wait_time):
        self.stages[stage_name] = {
        'processing_time': processing_time,
        'wait_time': wait_time
        }

# Example usage
product_1 = Product("Product_1")
product_1.add_stage_data("Assembly", processing_time=12, wait_time=3)
product_1.add_stage_data("Packaging", processing_time=8, wait_time=1)
```

This structure allows you to neatly organise data by product and stage, making it easier to track and access later. Each product can be treated as a self-contained unit with its own data, which can be further processed or analysed at the end of the simulation.

Event Tracing

Event tracing is an advanced data collection method that records the exact flow of a simulation by logging specific events. This allows for granular analysis of how entities interact with each other and the environment. Tracing specific events—such as the arrival and departure of entities, or resource requests and releases—can help identify bottlenecks, performance issues, or unexpected behaviours within a system.

Recording Event Timestamps and Types

To implement event tracing, you can log details about each event that occurs, including timestamps, event types, and the entities involved. This data can later be used for debugging or for deeper analysis of system dynamics.

```
class EventTracer:
        def __init__(self):
        self.events = []

        def log_event(self, event_type, entity_name, timestamp):
        self.events.append({
        'event_type': event_type,
        'entity_name': entity_name,
        'timestamp': timestamp
        })

# Example usage within a process
tracer = EventTracer()

def process(env, tracer, entity_name):
        arrival_time = env.now
        tracer.log_event("Arrival", entity_name, arrival_time)
        yield env.timeout(5)  # Simulate processing time
        departure_time = env.now
        tracer.log_event("Departure", entity_name, departure_time)
```

By keeping a detailed event log, you can examine the sequence of events after the simulation completes and assess how efficiently different parts of the system performed.

Data Storage Options

As simulations generate more data, you will need efficient ways to store and manage that information. Depending on the complexity and size of your dataset, you may choose between in-memory storage, file-based storage, or database solutions.

In-Memory Storage

For smaller simulations, storing data in memory using lists, dictionaries, or custom objects can be sufficient. This approach is fast and allows for easy data manipulation during the simulation. However, for larger datasets or long-running simulations, this method may become impractical due to memory limitations.

File-Based Storage

Storing data in files is a more scalable solution for simulations that generate large amounts of data. Common file formats include CSV, JSON, and HDF5. Libraries like pandas simplify the process of saving, loading, and managing large datasets.

Example: Storing Data with Pandas

```
import pandas as pd

# Create a DataFrame with simulation data
data = {
        'Stage': ['Assembly', 'Packaging'],
        'Processing Time': [12, 8],
        'Wait Time': [3, 1]
}
df = pd.DataFrame(data)

# Save DataFrame to a CSV file
df.to_csv('simulation_data.csv', index=False)

# Load the data back into a DataFrame
loaded_data = pd.read_csv('simulation_data.csv')
```

This approach is particularly useful for saving structured data that can later be analysed using tools like Excel or Python's data manipulation libraries.

Database Storage

For large-scale simulations that require fast access to vast amounts of data, databases offer a robust solution. SQL databases (e.g., MySQL, PostgreSQL) and NoSQL databases (e.g., MongoDB) are commonly used to store simulation results in a way that allows for quick querying and retrieval. This is especially useful when working in multi-user environments or when simulation data needs to be accessed and analysed over time.

Example: Storing Data with SQLite

```
import sqlite3

# Create a connection to the database
conn = sqlite3.connect('simulation_results.db')
cursor = conn.cursor()

# Create a table to store simulation data
cursor.execute('''CREATE TABLE IF NOT EXISTS simulation_data
```

```
(entity_name TEXT, event_type TEXT, timestamp REAL)"')

# Insert data into the table
cursor.execute("INSERT INTO simulation_data VALUES ('Entity_1', 'Arrival', 10.5)")

# Commit and close the connection
conn.commit()
conn.close()
```

Using a database allows for greater flexibility when handling large datasets, and it enables more advanced querying options to explore your simulation data efficiently.

Leveraging Pandas for Data Handling

Pandas is an essential library for handling simulation data in Python. It provides powerful tools for working with structured datasets, including filtering, aggregation, and joining data from multiple sources. Pandas simplifies many common data manipulation tasks.

Example: Filtering and Aggregating Data with Pandas

```
import pandas as pd

# Load simulation data from a CSV file
df = pd.read_csv('simulation_data.csv')

# Filter data to show only assembly stages
assembly_data = df[df['Stage'] == 'Assembly']

# Calculate the average processing time for assembly stages
avg_processing_time = assembly_data['Processing Time'].mean()
print(f"Average Assembly Processing Time: {avg_processing_time}")
```

By using pandas, you can perform complex data manipulations and analysis with minimal code, making it an indispensable tool for processing simulation results.

The advanced data collection techniques outlined in this section provide the foundation for effective simulation analysis. By organising data using custom structures, implementing event tracing, and selecting appropriate storage solutions, you'll be well-prepared to manage even the most complex datasets your simulations produce.

In the next section, we'll explore how to analyse the data you've collected, using descriptive and inferential statistical methods to extract meaningful insights from your simulations.

Statistical Analysis of Simulation Data

Once you have collected and stored your simulation data, the next step is to analyse it. Statistical analysis enables you to summarise the data, identify trends, and make informed decisions based on your simulation's performance. In this section, we'll dive deeper into both descriptive and inferential statistics, helping you extract actionable insights from your simulation data.

Descriptive Statistics

Descriptive statistics provide a simple way to summarise your simulation data by calculating key metrics such as the mean, median, variance, and percentiles. These statistics are critical for understanding the overall performance of your system and can highlight areas that require further investigation.

Measures of Central Tendency

The mean (average) and median (middle value) are two of the most commonly used metrics to understand the central tendency of your simulation data. These measures provide insights into the typical performance of your system.

```
# Calculate mean and median processing time
mean_processing_time = df['Processing Time'].mean()
median_processing_time = df['Processing Time'].median()

print(f"Mean Processing Time: {mean_processing_time}")
print(f"Median Processing Time: {median_processing_time}")
```

The mean is useful for understanding the average performance, but it can be skewed by outliers. In contrast, the median offers a better indication of typical performance in systems where outliers are present.

Measures of Variability

In addition to central tendency, it's important to understand the variability in your system's performance. Metrics like variance and standard deviation provide insights into the spread of your data, helping you assess how consistent your system is.

```
# Calculate variance and standard deviation of processing times
variance = df['Processing Time'].var()
```

```
std_deviation = df['Processing Time'].std()

print(f"Variance: {variance}")
print(f"Standard Deviation: {std_deviation}")
```

High variance or standard deviation may indicate that there are significant fluctuations in the performance of the system, which could be a sign of inefficiency or instability.

Percentiles

Percentiles are useful for understanding the distribution of data and identifying outliers. For example, the 90th percentile can show you the threshold under which 90% of the system's performance falls.

```
# Calculate the 90th percentile of processing times
percentile_90 = df['Processing Time'].quantile(0.9)
print(f"90th Percentile Processing Time: {percentile_90}")
```

This helps identify extreme cases where certain processes take significantly longer than usual, allowing you to investigate potential bottlenecks or inefficiencies in the system.

Probability Distributions

In many simulations, the data you collect may follow a particular probability distribution. Understanding these distributions is critical for interpreting your data and applying statistical models correctly.

Common Distributions in Simulations

- Normal Distribution: Often used when data is symmetrically distributed around a mean.
- Exponential Distribution: Common in queuing systems, where the time between arrivals or service times is random but follows a known rate.
- Triangular Distribution: Fantastic for modelling based on stakeholder information: you can gather a "best case", a "worst case" ad a "typical case" through stakeholders interviews to form the distribution.

Fitting Data to a Distribution

Fitting your simulation data to a probability distribution helps you understand the underlying dynamics of the system. Libraries like SciPy provide tools to fit data to different distributions.

```
from scipy.stats import expon, norm

# Fit data to an exponential distribution
exp_params = expon.fit(df['Processing Time'])

# Fit data to a normal distribution
norm_params = norm.fit(df['Processing Time'])

print(f"Exponential Distribution Parameters: {exp_params}")
print(f"Normal Distribution Parameters: {norm_params}")
```

By fitting the data to these distributions, you can determine which model best describes the behaviour of your system. This is particularly useful for making predictions about future performance or simulating different scenarios based on historical data.

Generating New Samples from the Fitted Model

After fitting the data to a distribution, you can draw samples from the fitted distribution to simulate new scenarios or extend your analysis. For example, if your data fits an exponential distribution well, you can generate new data points that mimic the behaviour of your system.

```
# Generate new samples from the fitted exponential distribution
new_exp_samples = expon.rvs(*exp_params, size=1000)

# Generate new samples from the fitted normal distribution
new_norm_samples = norm.rvs(*norm_params, size=1000)

print(f"First 10 samples from Exponential Distribution: {new_exp_samples[:10]}")
print(f"First 10 samples from Normal Distribution: {new_norm_samples[:10]}")
```

In this example, `expon.rvs()` and `norm.rvs()` generate 1,000 new random samples based on the fitted exponential and normal distributions, respectively. These newly generated data points can be used in further simulations to explore how your system might perform under similar conditions in the future.

Applications of Drawing New Samples

Generating new data samples from fitted distributions has several practical applications:

- Scenario Testing: Simulate different scenarios based on real-world data and assess system performance under various conditions.

- Monte Carlo Simulations: Use randomly generated samples from your fitted model to run multiple simulation iterations, allowing you to quantify risk and uncertainty in your system.
- Stress Testing: By generating extreme values from the fitted distribution, you can evaluate how your system behaves under high-stress or rare events, such as peak demand periods or service interruptions.

This method is an essential tool in both validating the robustness of your simulation and preparing for unexpected changes in system behaviour.

By fitting distributions to your data and generating new samples, you can create more realistic and data-driven simulations. This technique allows you to explore future scenarios with a high degree of confidence, ensuring that your decisions are based on statistically sound predictions.

Confidence Intervals

A confidence interval provides a range of values within which we can expect the true mean of the population to lie. This is particularly useful when comparing different scenarios or systems, as it provides an estimate of the uncertainty in the results.

```
import scipy.stats as stats

# Calculate 95% confidence interval for the mean
mean = df['Processing Time'].mean()
std_error = df['Processing Time'].sem()  # Standard error of the mean
confidence_interval = stats.t.interval(0.95, len(df)-1, loc=mean, scale=std_error)

print(f"95% Confidence Interval for the Mean: {confidence_interval}")
```

Confidence intervals provide a more complete understanding of your data by accounting for uncertainty, making them invaluable when presenting results or making decisions based on simulation data.

Hypothesis Testing

In simulations, hypothesis testing allows you to compare the performance of different scenarios and determine whether the observed differences are statistically significant. For example, you may want to test whether changing a parameter in your system (e.g., increasing the number of machines) leads to a significant improvement in processing times.

One of the most common tests is the t-test, which compares the means of two groups to determine whether they are significantly different.

```
# Perform a t-test to compare processing times between two scenarios
scenario_a = df[df['Scenario'] == 'A']['Processing Time']
scenario_b = df[df['Scenario'] == 'B']['Processing Time']
t_stat, p_value = stats.ttest_ind(scenario_a, scenario_b)

print(f"T-statistic: {t_stat}, P-value: {p_value}")
```

If the p-value is below a certain threshold (commonly 0.05), you can conclude that there is a statistically significant difference between the two scenarios.

Comparing Simulation Results Under Different Scenarios

One of the most powerful uses of inferential statistics is comparing simulation results under different scenarios. By running simulations with varying parameters, you can test which configurations perform better and make data-driven decisions based on the statistical analysis of the results.

Example: Comparing Two Resource Allocation Strategies
Imagine you're simulating two different resource allocation strategies in a factory: one where workers are assigned to specific machines (Scenario A) and one where they float between machines based on demand (Scenario B). Using hypothesis testing, you can compare the average processing times under each scenario and determine which strategy is more efficient.

```
scenario_a = df[df['Scenario'] == 'A']['Processing Time']
scenario_b = df[df['Scenario'] == 'B']['Processing Time']

# Perform a t-test to compare the means
t_stat, p_value = stats.ttest_ind(scenario_a, scenario_b)

if p_value < 0.05:
        print("Significant difference between the two strategies.")
else:
        print("No significant difference between the two strategies.")
```

By using statistical methods like hypothesis testing and confidence intervals, you can confidently choose the optimal strategy based on data rather than intuition.

Statistical analysis is an essential step in understanding your simulation data. By applying descriptive statistics, fitting data to probability distributions, and using inferential statistics, you can derive meaningful insights and make informed decisions based on the results of your simulations. In the next section, we'll explore advanced data visualisation techniques to help you better understand and communicate your simulation results.

Advanced Data Visualisation

While basic plotting libraries like Matplotlib offer core functionality for creating charts, more advanced libraries such as Seaborn and Plotly provide additional features that enhance your ability to present data effectively.

Seaborn for Sophisticated Data Visualisations

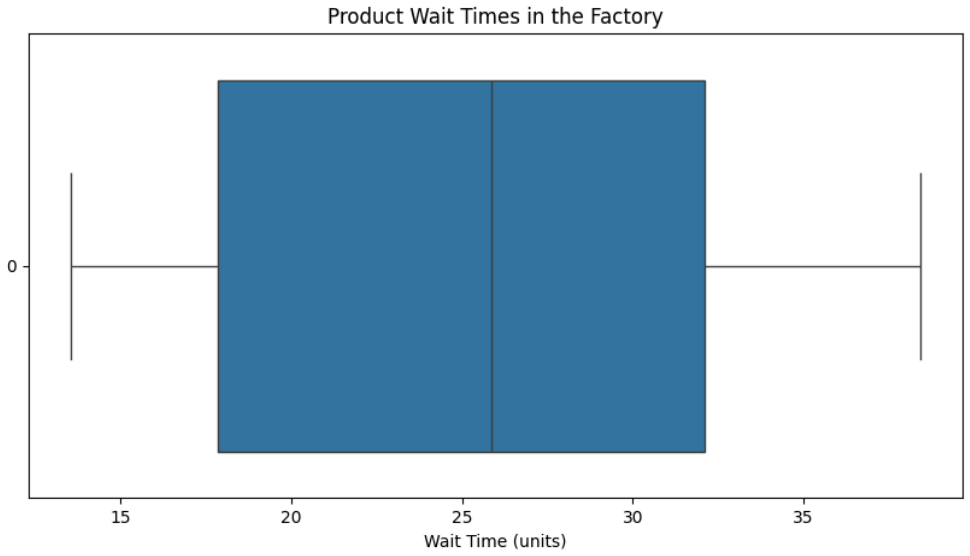
Seaborn is a Python library built on top of Matplotlib, designed for making complex visualisations simpler. It allows for cleaner, more informative plots that are especially useful when analysing data distributions and relationships between variables.

Example: Creating a Box Plot and Violin Plot

A box plot summarises the distribution of a dataset, showing the median, quartiles, and outliers. A violin plot goes a step further by illustrating the density of the data, providing a richer visualisation of the distribution.

```
import seaborn as sns
import matplotlib.pyplot as plt

# Box plot for processing times by stage
sns.boxplot(x='Stage', y='Processing Time', data=df)
plt.title('Box Plot of Processing Times by Stage')
plt.show()
```

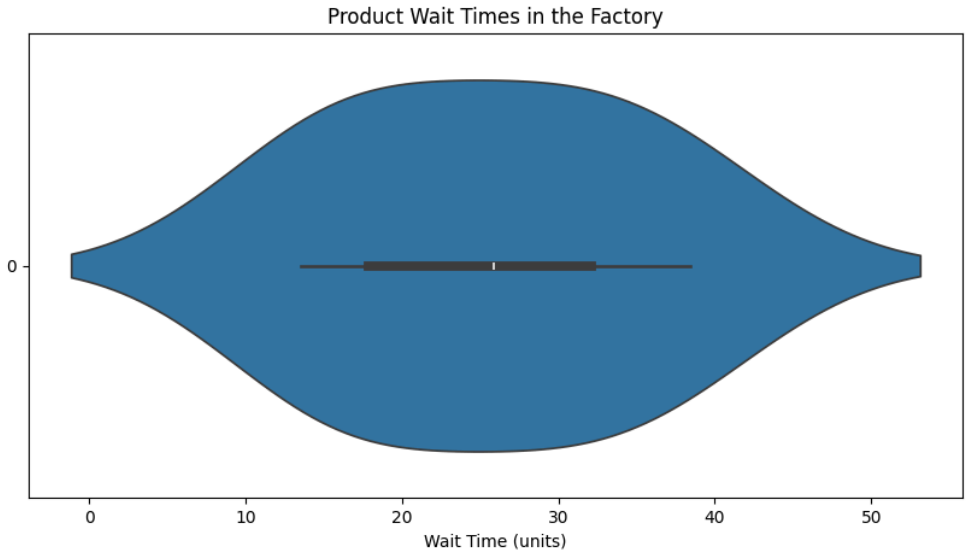
School of Simulation



Here's what each part of the box plot represents:

- **Central Line (Median)**: Represents the median (middle value) of the dataset. Half of the data lies above this line, and half below.
- **Box (Interquartile Range - IQR)**: The box spans from the first quartile (Q1, the 25th percentile) to the third quartile (Q3, the 75th percentile). This range covers the middle 50% of the data.
- **Whiskers**: Extend from the box to the smallest and largest values within 1.5 × IQR from the edges of the box. These lines illustrate the expected range of typical data values, excluding outliers.
- **Outliers**: Shown as individual points beyond the whiskers. These are data points significantly different from other observations.

```
# Violin plot for visualising the distribution of processing times
sns.violinplot(x='Stage', y='Processing Time', data=df)
plt.title('Violin Plot of Processing Times by Stage')
plt.show()
```



Here's what a violin plot represents:

- **Width (Density)**: The width at any given point represents the **density** (frequency) of the data at that value. A wider section means more data points are concentrated there, while narrower sections indicate fewer observations.
- **Central Marker (Median or Mean)**: Often shown as a dot or a small line inside the violin, this represents the median or mean of the dataset.
- **Box and Whisker Inside the Violin** *(Optional but commonly included in Seaborn)*:
- **Box**: Represents the interquartile range (25th to 75th percentile, Q1 to Q3).
- **Whiskers**: Extend to the largest and smallest values within 1.5 × IQR from the box edges, showing typical data spread.
- **Outliers**: Sometimes shown as points beyond whiskers (optional).

Box and violin plots are particularly helpful for comparing multiple groups in your simulation, such as different stages in a production line or various configurations in a logistics simulation.

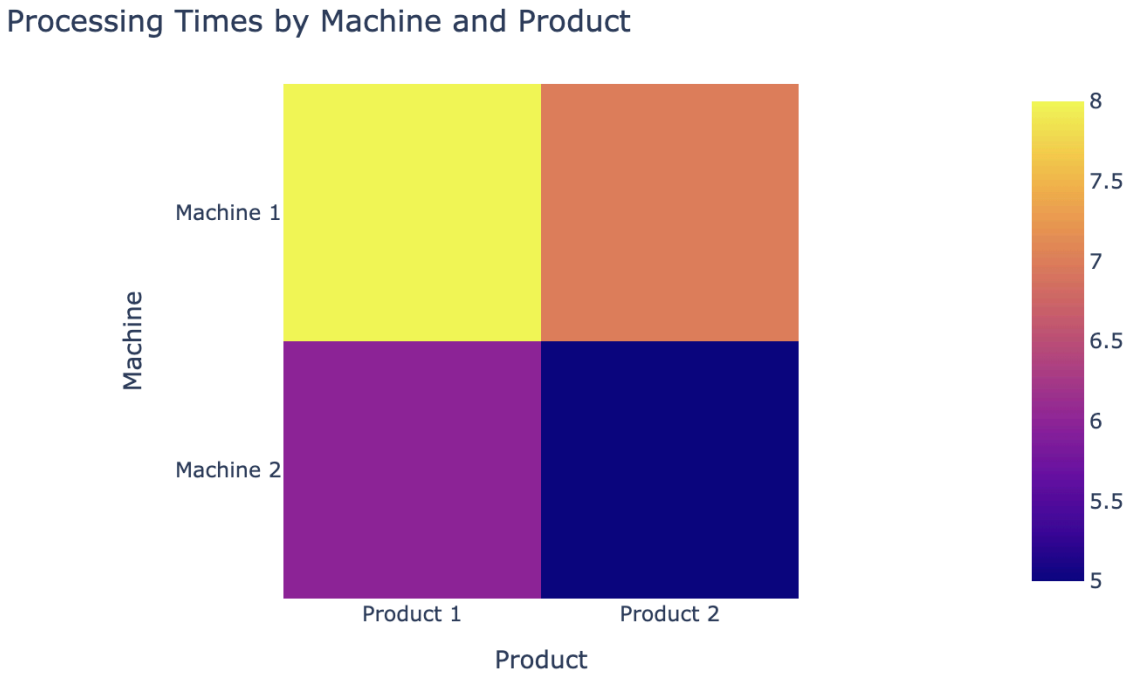
Plotly for Interactive Visualisations

Plotly is a powerful library for creating interactive, web-based visualisations. It supports a wide variety of plots and charts and allows users to zoom in, hover over data points, and explore data in more detail. Plotly is especially useful when you need to share your results with others or when exploring large datasets.

Example: Creating an Interactive Heatmap

A heatmap is a useful tool for visualising the intensity of data across two dimensions. In the context of simulations, it can help you understand correlations between variables or highlight areas where performance issues may arise.

```
import plotly.express as px

data = {
        'Machine': ['Machine 1', 'Machine 1', 'Machine 2', 'Machine 2'],
        'Product': ['Product 1', 'Product 2', 'Product 1', 'Product 2'],
        'Processing Time': [8, 7, 6, 5]
}
df = pd.DataFrame(data)
fig = px.imshow(df.pivot(index='Machine', columns='Product', values='Processing Time'))
fig.update_layout(title='Processing Times by Machine and Product')
fig.show()
```



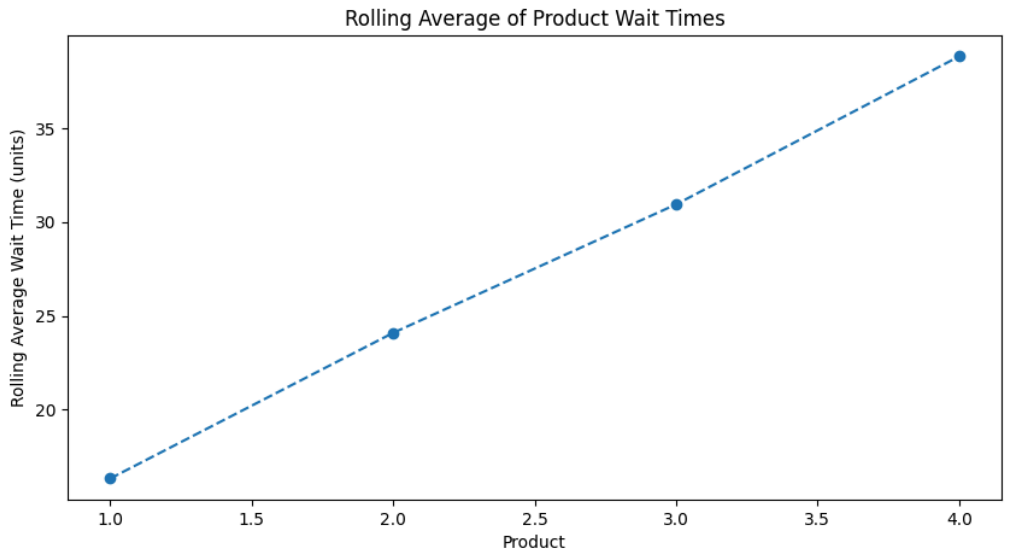
By using Plotly, you can create highly interactive visualisations that allow for deeper exploration of the data. This is ideal for collaborative analysis or for presentations where stakeholders need to interact with the data themselves.

Time-Series Analysis

Simulations often generate time-based data, such as the time required for entities to pass through various stages or the waiting times in a queue. Time-series analysis allows you to visualise these trends over time, helping you identify patterns, peaks, and potential inefficiencies.

Example: Visualising Wait Times Over Time
Using rolling statistics or moving averages, you can smooth out short-term fluctuations and highlight longer-term trends in your data.

```
# Calculate rolling average of wait times over time
rolling_average = pd.Series(product_wait_times).rolling(window=2).mean()

# Plot the rolling average
plt.figure(figsize=(10, 5))
plt.plot(rolling_average, marker='o', linestyle='--')
plt.title('Rolling Average of Product Wait Times')
plt.xlabel('Product')
plt.ylabel('Rolling Average Wait Time (units)')
plt.show()
```



This type of visualisation is particularly useful in queueing simulations or in systems where performance varies significantly over time, such as traffic flow or patient arrivals at a hospital.

School of Simulation

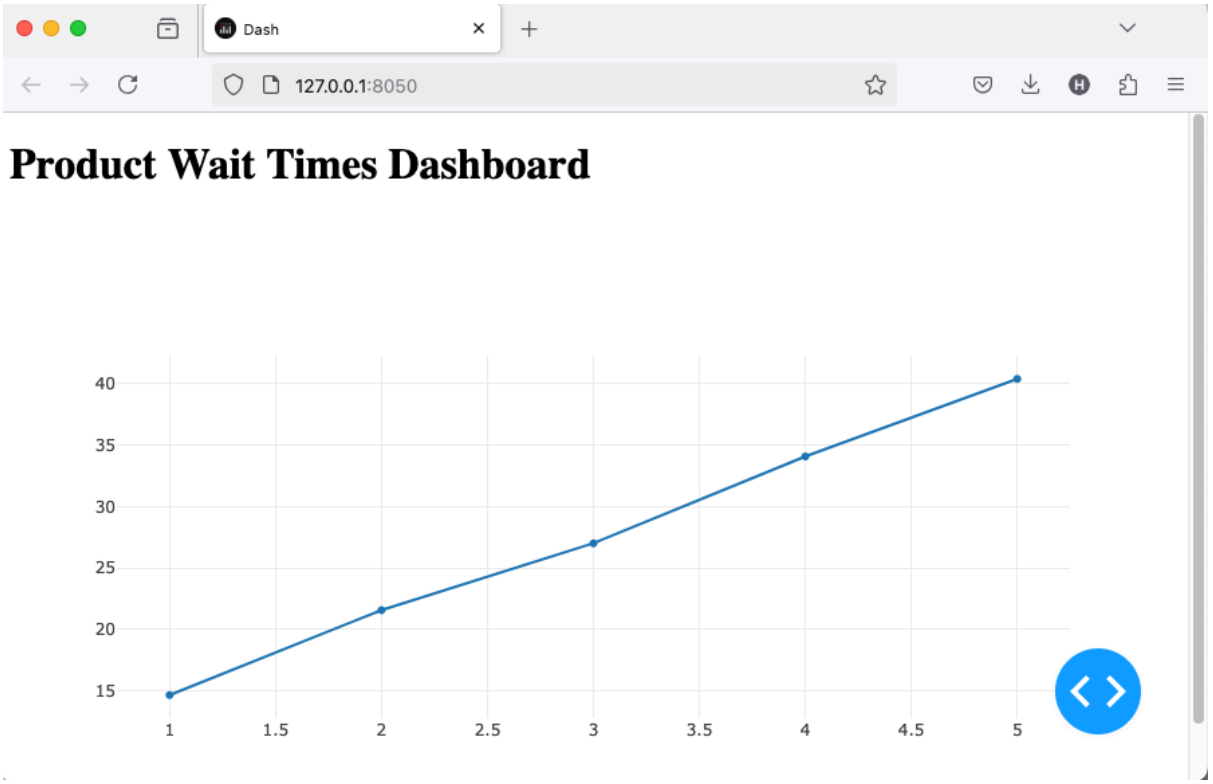
Interactive Dashboards

Interactive dashboards allow you to dynamically explore your simulation data, providing an intuitive way to filter, drill down, and compare different aspects of your results. Tools like Dash (built on top of Plotly) and Bokeh are ideal for creating web-based, interactive dashboards that can be shared with others for further analysis.

Example: Creating a Simple Dashboard with Dash
Dash is a Python framework for building analytical web applications. You can create interactive dashboards that allow users to manipulate and visualise simulation data in real-time.

```
import dash
import dash_core_components as dcc
from dash import html

# Create a Dash application
app = dash.Dash(__name__)

# Define the layout of the dashboard
app.layout = html.Div([
        html.H1('Product Wait Times Dashboard'),
        dcc.Graph(
        id='product-wait-times',
        figure={
        'data': [{
                'x': list(range(1, len(product_wait_times) + 1)),
                'y': product_wait_times,
                'type': 'line',
                'name': 'Wait Time'
        }],
        'layout': {
                'title': 'Product Wait Times',
                'xaxis': {'title': 'Product'},
                'yaxis': {'title': 'Wait Time (units)'}
        }
        }
        )
])

# Run the Dash application
if __name__ == '__main__':
        app.run_server(debug=True)
```

School of Simulation



With Dash, you can build a custom dashboard that visualises key metrics from your simulation, such as processing times, resource utilisation, or throughput. Interactive features like sliders, drop-down menus, and real-time updates make it easy to explore different scenarios or drill down into specific aspects of the simulation.

Benefits of Interactive Visualisations

Interactive visualisations, such as those created with Plotly and Dash, offer several advantages:

- Exploratory Data Analysis: By allowing users to interact with the data, these visualisations make it easier to identify trends, correlations, and outliers that might be missed in static charts.
- Collaboration: Interactive dashboards can be shared with team members or stakeholders, facilitating collaborative decision-making based on simulation results.
- Dynamic Exploration: Users can manipulate filters, adjust parameters, and zoom in on specific areas of interest, providing a more engaging and insightful analysis experience.

Conclusion

Effectively analysing and visualising simulation data is vital if you want to truly understand how your systems are performing and make informed decisions. In this chapter, we've explored a variety of advanced techniques to elevate your approach to simulation analysis. You've learned to manage and structure complex data sets, apply inferential statistics to draw meaningful conclusions, and even predict future system behaviours. Essentially, you've turned raw numbers into practical, actionable insights.

We also covered advanced visualisation techniques using powerful libraries like Seaborn and Plotly, helping you present your data clearly and convincingly. Whether through compelling static visuals or interactive dashboards, you now have the tools to effortlessly communicate trends, compare different scenarios, and emphasise key performance indicators.

By integrating effective data collection methods, rigorous statistical analysis, and engaging visualisation strategies, you're well-equipped to deeply understand your simulation results. This will help you optimise processes, anticipate how changes could impact performance, and ultimately support smarter, data-driven decision-making.

As you apply these skills in your own simulations, remember: the real power of analysis lies not merely in collecting data, but in translating it into meaningful insights that genuinely improve your systems.

Monte Carlo Simulation and Full Factorial Analysis

Now that you have a solid understanding of building and analysing simulations, it's time to delve into some more advanced topics. These will push the boundaries of what you can achieve with SimPy and open the door to even more sophisticated simulations.

Simulate uncertainty with Monte Carlo methods or conduct a full factorial analysis - these techniques will allow you to handle complex systems and draw deeper insights from your models.

Introduction to Monte Carlo Simulations

Monte Carlo simulations are a powerful tool used to model uncertainty in systems. They are particularly useful when you're dealing with processes that involve randomness or variability, such as fluctuating demand in a supply chain or uncertain lead times in project management.

By running a simulation many times with different random inputs, you can create a distribution of possible outcomes, helping you understand the range of scenarios your system might encounter.

How Monte Carlo Simulations Work

In a Monte Carlo simulation, you repeatedly run the same simulation with random variations in key parameters. These parameters could be anything from customer arrival times to machine breakdowns. By running the simulation multiple times with different random inputs, you generate a distribution of possible outcomes, allowing you to calculate probabilities, averages, and confidence intervals.

For example, let's say you're managing a warehouse and you want to simulate the time it takes to fulfil orders. Some orders may be fulfilled quickly, while others take longer due to stock shortages or shipping delays. A Monte Carlo simulation would allow you to model this variability by introducing randomness into the process times and running the simulation thousands of times to see the distribution of fulfilment times.

Example: A Monte Carlo Simulation in SimPy

Let's build a simple Monte Carlo simulation of a factory with randomised processing times. We'll run the simulation multiple times, each with different random input values, to see the distribution of total factory processing times.

```
import simpy
import random

# Data collection for multiple runs
monte_carlo_results = []

# Factory process with randomised processing times
def factory_process(env, machine):
        start_time = env.now

        # Machine processing with random times
        with machine.request() as req:
        yield req
        yield env.timeout(random.uniform(5, 15))  # Random processing time between 5 and
15 units

        total_time = env.now - start_time
        return total_time

# Monte Carlo simulation runner
def run_monte_carlo(env, machine, num_runs):
        for _ in range(num_runs):
        total_time = yield env.process(factory_process(env, machine))
        monte_carlo_results.append(total_time)

# Set up the simulation environment
env = simpy.Environment()

# Create a machine resource
machine = simpy.Resource(env, capacity=1)

# Run Monte Carlo simulation with 100 runs
env.process(run_monte_carlo(env, machine, num_runs=100))
env.run()

# Output the distribution of results
print(f'Average factory processing time: {sum(monte_carlo_results) /
len(monte_carlo_results):.2f}')
```

Output:
Average factory processing time: 10.28

Interpreting the Results

After running the simulation 100 times, you can calculate and analyse the distribution of total processing times. This will give you insights into the likely range of outcomes for your factory's performance, allowing you to plan for uncertainty.

For instance, the output might show that the average processing time is 10.2 units, but with a range that spans from 5 to 15 units. This tells you that, while the system usually performs within a certain range, there are cases where processing times might be significantly longer or shorter. With this information, you can make better decisions about resource allocation, scheduling, and risk management.

Benefits of Monte Carlo Simulations

Monte Carlo simulations are particularly useful in situations where:

- Uncertainty and Variability: When you're dealing with processes that involve randomness, Monte Carlo methods allow you to simulate a wide range of possible outcomes.
- Risk Assessment: By running the simulation multiple times, you can assess the likelihood of different scenarios, helping you quantify risks and prepare for unexpected events.
- Decision Support: Monte Carlo simulations allow you to test different strategies under uncertainty. For example, you could experiment with different resource capacities or scheduling policies to see how they impact system performance in the face of random variability.

Implementing Monte Carlo Simulations in SimPy

SimPy's flexibility makes it a great tool for implementing Monte Carlo simulations. By adding random elements to the timing of events, you can simulate variability and uncertainty in real-world processes.

Here's how to implement a Monte Carlo simulation in SimPy step by step:

1. Introduce Randomness in Processes

Use Python's random module to introduce randomness into your processes. This could be random customer arrival times, processing times, or resource availability. For example, you might simulate customers arriving at random intervals by using random.expovariate() to model exponentially distributed inter-arrival times.

```
arrival_time = random.expovariate(1/5)  # Mean arrival time of 5 units
```

2. Run the Simulation Multiple Times

A key feature of Monte Carlo simulations is that they are run multiple times, each with different random inputs. In SimPy, this is easily done by running the simulation in a loop, resetting the environment each time.

Full Factorial Analysis

A full factorial analysis is a method used to evaluate all possible combinations of input parameters in a simulation. This approach is particularly useful when you want to understand how different factors interact and impact the system's performance. For example, in a manufacturing simulation, you might vary the number of machines, the processing time, and the arrival rate of products, and evaluate all combinations to see how they affect throughput and wait times.

Example: Full Factorial Analysis in a Factory Simulation

Let's extend our factory example to conduct a full factorial analysis. We'll vary the capacity of the machines and the processing times to see how these factors impact the system's performance.

```
import simpy
import numpy as np
import pandas as pd
import itertools


def job(env, job_id, server, arrival_time, service_time, data):
    """
    A single job process:
    - Arrives at 'env.now' which is recorded as 'arrival_time'.
    - Requests the server resource.
    - Waits until it can be served.
    - After obtaining the server, 'works' for an exponential time.
    - Records waiting time and completion time in 'data'.
    """
    # Mark the time the job actually arrived in the system.
    # (We store it here in case the job gets queued or if there's a waiting line.)
```

```
        arr_time = arrival_time

        # Request the server
        with server.request() as req:
        yield req
        start_service_time = env.now
        waiting_time = start_service_time - arr_time

        # "Work" (service time is exponential with mean = service_time param)
        yield env.timeout(np.random.exponential(service_time))

        completion_time = env.now

        # Record relevant information about this job
        data["job_id"].append(job_id)
        data["arrival_time"].append(arr_time)
        data["start_service_time"].append(start_service_time)
        data["waiting_time"].append(waiting_time)
        data["completion_time"].append(completion_time)


def arrival_process(env, arrival_rate, service_time, server, capacity, data):
        """
        A process that spawns jobs according to a Poisson process (exponential
        inter-arrival times). If the queue (and resource) is at capacity, the job is lost.
        """
        job_counter = 0
        while True:
        # Wait for the next arrival
        inter_arrival = np.random.exponential(1.0 / arrival_rate)
        yield env.timeout(inter_arrival)

        job_counter += 1
        arrival_time = env.now

        # Check if there's space in the queue
        # "server.count" = how many are currently being served
        # "len(server.queue)" = how many are waiting
        if server.count + len(server.queue) < capacity:
        # Start a job process
        env.process(job(env, job_counter, server, arrival_time, service_time, data))
        else:
        # If at capacity, we can record that a job was lost (optional)
```

```
            data["lost_jobs"] += 1


def run_simulation(arrival_rate, service_time, queue_capacity, sim_time=50):
            """
            Sets up and runs the simulation with the given parameters, returns summary metrics.
            """
            # Create environment
            env = simpy.Environment()

            # Define a single server resource with capacity for 1 in service at a time
            server = simpy.Resource(env, capacity=1)

            # Create a structure to store event-level data
            data = {
            "job_id": [],
            "arrival_time": [],
            "start_service_time": [],
            "waiting_time": [],
            "completion_time": [],
            "lost_jobs": 0  # Keep track of how many jobs arrived but couldn't join queue
            }

            # Start arrival process
            env.process(arrival_process(env, arrival_rate, service_time, server, queue_capacity,
data))

            # Run the simulation
            env.run(until=sim_time)

            # Once done, summarise key metrics
            completed_jobs = len(data["job_id"])  # Number of jobs actually served
            throughput = completed_jobs / sim_time  # jobs per unit time

            if completed_jobs > 0:
            avg_waiting_time = np.mean(data["waiting_time"])
            # Time in system = completion_time - arrival_time
            times_in_system = [c - a for c, a in zip(data["completion_time"], data["arrival_time"])]
            avg_time_in_system = np.mean(times_in_system)
            else:
            avg_waiting_time = 0
            avg_time_in_system = 0
```

```
        return {
        "arrival_rate": arrival_rate,
        "service_time": service_time,
        "queue_capacity": queue_capacity,
        "throughput": throughput,
        "avg_waiting_time": avg_waiting_time,
        "avg_time_in_system": avg_time_in_system,
        "lost_jobs": data["lost_jobs"]
        }


# Let's define a few levels for each parameter
arrival_rates = [0.8, 1.0, 1.2]
service_times = [1.0, 1.5]
queue_capacities = [2, 5]

# Keep simulation time short for demonstration
SIM_TIME = 50

# Run all combinations
results = []
for arr_rate, svc_time, cap in itertools.product(arrival_rates, service_times,
queue_capacities):
        outcome = run_simulation(arr_rate, svc_time, cap, sim_time=SIM_TIME)
        results.append(outcome)

# Put into a DataFrame
df = pd.DataFrame(results)
print("Experiment Results:")
print(df)

# Simple Multivariate Analysis
correlation_matrix = df.corr(numeric_only=True)
print("\nCorrelation Matrix:")
print(correlation_matrix)
```

Output:

Experiment Results:

	arrival_rate	service_time	...	avg_time_in_system	lost_jobs
0	0.8	1.0	...	1.793195	12
1	0.8	1.0	...	1.116539	0
2	0.8	1.5	...	2.226897	15
3	0.8	1.5	...	4.766826	16
4	1.0	1.0	...	1.491335	18

5	1.0	1.0	...	2.129484	4
6	1.0	1.5	...	2.000232	19
7	1.0	1.5	...	3.887218	1
8	1.2	1.0	...	1.500817	17
9	1.2	1.0	...	3.603226	15
10	1.2	1.5	...	2.988851	43
11	1.2	1.5	...	5.349550	34

[12 rows x 7 columns]

Correlation Matrix:

```
Correlation Matrix:
                    arrival_rate  service_time  ...  avg_time_in_system  lost_jobs
arrival_rate        1.000000e+00 -2.691152e-16  ...            0.274013   0.564343
service_time       -2.691152e-16  1.000000e+00  ...            0.605951   0.432858
queue_capacity     -9.442639e-17 -6.167906e-17  ...            0.559582  -0.377005
throughput          4.330127e-01 -5.474375e-01  ...            0.031041  -0.297477
avg_waiting_time    2.740879e-01  4.794861e-01  ...            0.981673   0.240979
avg_time_in_system  2.740127e-01  6.059507e-01  ...            1.000000   0.356288
lost_jobs           5.643431e-01  4.328579e-01  ...            0.356288   1.000000
```

Correlation Matrix

The **correlation matrix** is a table that shows us the pairwise linear relationships between all numerical variables in your dataset. In the final code example, we have columns such as:

- arrival_rate
- service_time
- queue_capacity
- throughput
- avg_waiting_time
- avg_time_in_system
- lost_jobs

When we call:
`correlation_matrix = df.corr(numeric_only=True)`

1. **Each row/column corresponds to one of these variables**.
2. **The numbers in the cells** are the **correlation coefficients** (often Pearson's r by default).
3. **Diagonal elements** (where row = column) are always **1**, because a variable is perfectly correlated with itself.

4. The matrix is **symmetric** about this diagonal.

How to interpret a correlation coefficient

- **1.0** = perfect positive linear relationship (if one variable goes up, the other goes up in lockstep).
- -**1.0** = perfect negative linear relationship (if one variable goes up, the other goes down in lockstep).
- **0.0** = no linear relationship (they vary independently in a linear sense, though they could still have a non-linear relationship).
- Values in between reflect weaker or moderate degrees of positive or negative association.

Why it is useful

- It gives a quick view of **which parameters have the strongest relationships** with each other.
- For example, you might see a **strong positive correlation** between **arrival_rate** and **avg_waiting_time** (because, as more jobs arrive, queues get longer and waiting times go up).
- You might see a **negative correlation** between **queue_capacity** and **lost_jobs** (increasing queue size means fewer dropped jobs).

Caveats

- **Correlation ≠ causation**. A high positive correlation doesn't prove that one variable causes another; it only signals that they tend to vary together in a linear way.
- **Non-linear patterns** can exist without showing up strongly in a simple correlation.

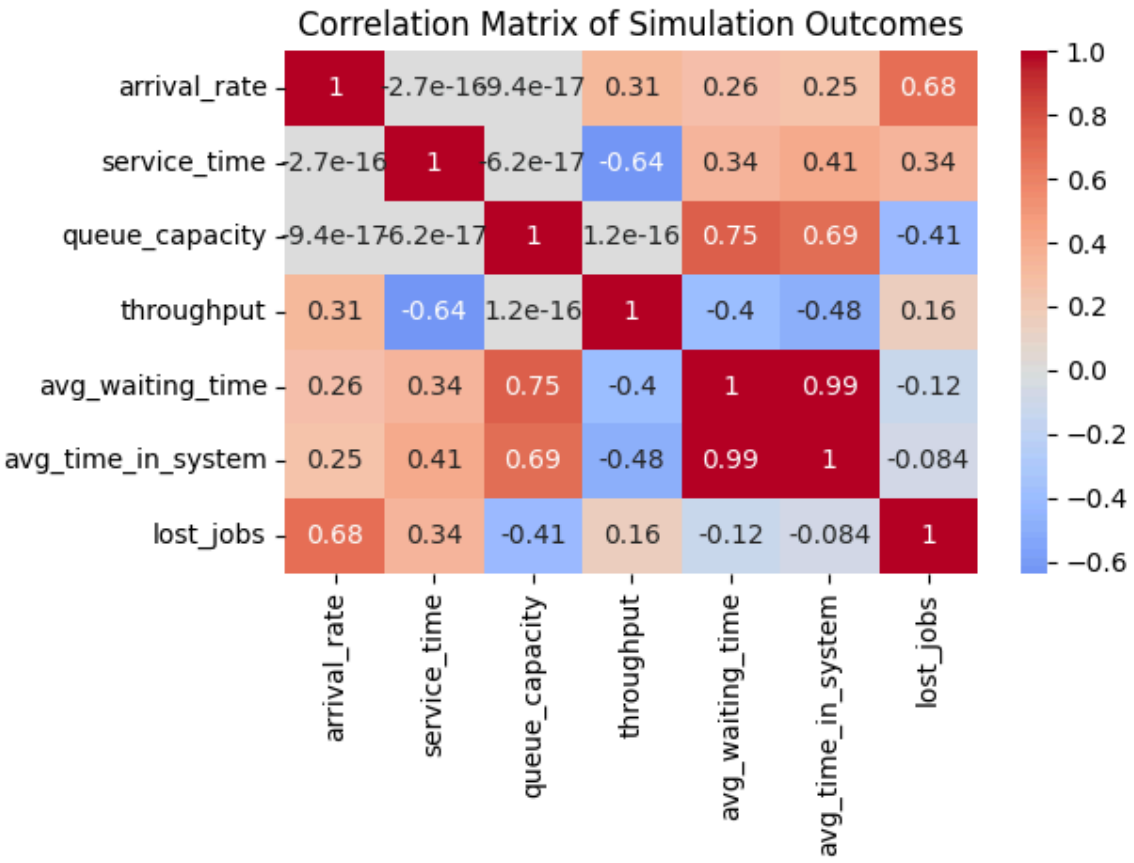
If your dataset is very small (few data points), correlation values might be unstable. For better statistical confidence, you'd typically run more simulation replications or gather more data points at each setting.

Visualising the Correlation Matrix

We can visualise the correlation matrix with a single Seaborn call:

```
import seaborn as sns
import matplotlib.pyplot as plt

sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', center=0)
```

```
plt.title("Correlation Matrix of Simulation Outcomes")
plt.tight_layout()
plt.show()
```



The correlation matrix is a powerful way of visualising the relationships at a glance between variables that make up your full factorial experiment.

Analysing the Results

After running the simulation for all combinations of machine capacities and processing times, you can analyse how these factors impact performance. A full factorial analysis provides insights into the interactions between variables and helps you identify the best configuration for optimising system performance.

For example, you might find that increasing the machine capacity has a much larger effect on reducing total processing time than changing the processing times, indicating that adding more machines is a more effective strategy for improving performance.

Benefits of Full Factorial Analysis

There are several key benefits of the full factorial analysis approach:

- Exploring All Combinations: Full factorial analysis allows you to explore every possible combination of input parameters, ensuring that you fully understand how different factors interact and affect system performance.
- Identifying Optimal Solutions: By analysing the results of each combination, you can identify the best configuration for your system and make informed decisions about resource allocation and process improvements.
- Understanding Interactions: Full factorial analysis highlights interactions between factors that might not be obvious from single-variable analysis. For example, you might discover that increasing machine capacity only improves performance when combined with faster processing times.

Both techniques allow you to push your simulations further, helping you gain deeper insights into system behaviour and make more informed decisions.

Classes vs Functions in SimPy

In SimPy, you can define processes either as standalone functions or as methods within classes. Both approaches work – SimPy treats them the same in terms of yielding events.

However, there are clear situations where using classes can make your simulation code more organised and extensible.

Complex Simulations with Multiple Entities

When modelling many similar objects (e.g. many machines, customers, vehicles), classes help encapsulate each entity's state and behaviour. Instead of juggling global variables for each entity, you create instances of a class, each holding its own data. This object-oriented approach mirrors real-world systems, making the model easier to understand

Processes Tied to Real-World Objects

If a process naturally belongs to an object (e.g. a Machine's operation, or a Customer's service process), it's logical to implement it as a method of a class representing that object. This ties the actions (SimPy process methods) to the data (attributes) they affect, following good encapsulation practices

Reuse and Extension

Functions are fine for simple scenarios, but classes shine when you want to extend functionality or reuse components. With classes, you can subclass or modify behaviour without rewriting the whole simulation. This is useful if requirements change or for building libraries of simulation components
For example, you might have a generic Machine class and later subclass it to a MachineWithBreakdown with additional failure logic.

Clarity and Maintenance

As simulations grow, purely functional code can become unwieldy, with many parameters passed around. Classes group related logic together (state + processes), making code more modular and easier to maintain. Each class has a clear responsibility (following the single responsibility principle), which improves readability.

On the other hand, for very simple simulations or quick prototypes, functions might suffice. If your simulation only involves one or two processes and minimal state, using classes could be overkill. But as a rule of thumb, once you have multiple instances of a similar entity or a need for structured organisation, it's time to use classes.

Classes are not required to use SimPy (SimPy 3+ allows generator functions for processes), but they become essential for managing complexity in larger simulations.

In summary, use classes when they model your problem's entities or make the design cleaner, and stick to simple functions only for the most trivial cases.

Best Practices for Modular Simulation Design

Designing a modular SimPy simulation means organising your code into independent, reusable components. Here are some best practices to achieve this.

Identify Entities and Processes

Begin by analysing the system and identifying the nouns (entities/objects) and verbs (processes/activities). Typically, each key entity in your real-world scenario can be a class in your simulation (e.g. machines, operators, customers), and their behaviours are methods (process-generators)
This OO mapping makes your code reflect the real system structure, aiding validation and understanding.

Encapsulate State in Classes

Give each class attributes to hold its state. For example, a Machine class might track whether it's busy or how many items it processed. By keeping state inside the object, you avoid global variables and make it clear which parts of the code manipulate which data.

Encapsulate Behaviour with Methods

Implement the entity's timeline of activities as a method (or multiple methods) using yield statements. These generator methods define what the entity does over time. This way, the logic for an entity is self-contained. For instance, a Truck class could have a drive() process method that handles its movement and deliveries.

Use an Initialisation Pattern

In SimPy, when you create an object that has a process, you typically start its process in the constructor (__init__). For example, env.process(self.run()) inside __init__ will immediately schedule that entity's process in the environment. This pattern ensures each entity begins its behaviour when created

(Remember, simply calling a generator function won't start it – you need to pass it to env.process() to get it scheduled.)

- Separate Concerns: Modular design often means separating different concerns of the simulation. For example, you might have:
- Classes for active entities (those that have behaviours over time, like machines or customers).
- Classes or simple data structures for passive resources (like simpy.Resource or a wrapper class if you need extended functionality).
- Functions or a class method to generate arrivals (e.g. a source generating new customers or parts periodically).
- A main section or class responsible for setting up the environment, creating initial objects, and running the simulation.This separation makes it easier to tweak one part (say, arrival rate or resource capacity) without affecting unrelated parts.

Parameterise Your Model

Avoid hard-coding numbers throughout the code. Define simulation parameters (like processing times, inter-arrival rates, number of machines) at the top of your script or in a config section. This makes your model easier to adjust and experiment with. For instance, set NUM_MACHINES = 5 or SERVICE_TIME = 8 in one place. This practice was used in the SimPy documentation's machine shop example where constants like processing time and repair time are defined at the top for easy tweaking

Reusability and Extensibility

Design classes that could be reused in similar simulations. Perhaps today you model one factory line; tomorrow you might model another with a few differences. If your classes are well-designed, you can extend them rather than starting from scratch. Object-oriented simulations allow development of libraries of components (for example, a library of manufacturing elements like machines, conveyors, operators) that can be reused in different models

Testing and Validation

Because your simulation is modular, you can test components in isolation. For example, you could test the process of a single machine by running a short simulation with one machine and ensuring it behaves as expected. Modular code is easier to debug since you can pinpoint which class or process might be responsible for any unexpected outcome.

By following these practices, you create simulations that are easier to read, share, and maintain. This modular approach also helps in explaining your model to stakeholders – each class or module corresponds to a real-world concept, which makes the model's logic more transparent.

Final Thoughts and the Next Discrete-Event

As we wrap up this book, now’s a good moment to pause and reflect - not just because you deserve a break, but also because applying what you've learned is key to making all this knowledge stick.

By this point, you've built a strong foundation in modelling and simulation using Python and SimPy. You can confidently create detailed simulations, manage resources effectively, and dig into data analysis to make informed decisions. But let's be honest: this is really just the beginning.

Reflecting on Your JourneyOver the course of this book, you've moved from grasping the basics of discrete-event simulation to mastering advanced topics like Monte Carlo simulations and factorial analysis.

You've learned how to:

- Build Simulations: Whether it's a straightforward queueing model or a complicated production process, you've got the hang of SimPy’s event-driven approach. You've brought together processes, resources, and events into realistic simulations-hopefully without too many headaches.
- Manage Resources: Efficient resource management can make or break a system. You've learned how to model everything from machines to people, spotting bottlenecks and opportunities to optimise along the way.
- Analyse Data: Building a simulation is only half the fun. You've also mastered interpreting metrics like wait times and resource utilisation, turning raw data into actionable insights - without falling asleep in a spreadsheet.
- Handle Uncertainty: Uncertainty isn't just inevitable; it's everywhere. With Monte Carlo simulations, you've practised introducing variability and stress-testing scenarios, so you're ready when reality throws curveballs.
- Test Scenarios: You've explored how to run 'what-if' scenarios, tweaking parameters to see how changes affect overall performance - without the risk of real-world disasters.The practical projects you've tackled have set you up nicely to create your own simulations, solve real-world problems, and apply these skills in your own field.

What’s Next?

If you've enjoyed getting stuck into simulations and want to keep pushing your skills further, there’s plenty more to explore. Here are a few areas worth diving into, all of which I cover in more depth in my School of Simulation course:

- Advanced Simulation Techniques: Go beyond the basics by mastering techniques like PriorityStores, FilterStores interrupts, and conditional events to tackle complex real-world challenges.
- Visual Modelling and Stakeholder Engagement: Learn how to transform stakeholder insights and requirements into actionable simulation models using visual mapping techniques - essential for getting your simulations right first time.
- Simulation Project Management: Building a great simulation is only part of the job. Develop the skills to scope, manage, and deliver high-impact projects, communicate effectively with stakeholders, and present results clearly.
- Optimisation and Automation: Use optimisation methods such as genetic algorithms and simulated annealing to fine-tune your models, ensuring they deliver maximum value with minimal fuss.
- Interactive and Animated Simulations: Discover how to bring your simulations to life with dynamic animations, making complex scenarios easy to understand and engaging to stakeholders.

Taking Simulation to the Real World

These skills aren't just theoretical - they're genuinely useful across industries, from manufacturing and healthcare to logistics and finance. Simulation helps you test ideas safely, optimise processes, and make decisions grounded in data, avoiding costly mistakes.

Here’s how you can practically start applying your new skills:
Start Small: Pick manageable projects to build confidence and finesse your skills. A small success beats a big, ambitious failure - every time.
Collaborate: Good simulations often involve teamwork. Work with colleagues or experts who can provide real-world context, better data, or just sanity checks.
Use Simulation for Decision-Making: Next time you're facing an important decision - like launching a new product or changing a process - let your simulations do the heavy lifting.
Keep Learning: Simulation is constantly evolving, so stay curious. Join online communities, attend conferences, or keep up with new research to stay ahead of the curve.

Final Thoughts

Remember, simulation isn't just numbers and algorithms. It's also an art. Combining technical know-how with a practical understanding of real-world systems makes you uniquely capable of solving challenging problems.

Thank you for taking this journey into simulation with Python and SimPy. Hopefully, this book has sparked your curiosity and given you the confidence to dive deeper into this fascinating

field. Every system is a potential simulation waiting to be built - and every simulation is a step toward a smarter solution.

If you fancy joining my free webinar, “How to Become a Go-To Expert in Simulation with Python,” you can register at: https://simulation.teachem.digital/webinar-signup

See you there!

P.s. if you found this guide helpful, I’d be so grateful for a review on Goodreads: https://www.goodreads.com/book/show/223221425-simulation-in-python-with-simpy

Thank you!

School of Simulation

About the Author

I've been working with simulation for over 14 years across all sorts of industries, from transport to mining to defence to energy.

Simulation is the beating heart of everything I do. It's how I 10x'd my annual earnings and achieved financial freedom. It's how I enjoy a fully flexible, remote lifestyle. And it's why people seek me out from all over the world for help with their modelling and simulation projects.

My work has never been dull: from individual contributor to team lead, tech lead, business owner and consultant. This allows me to bring a unique perspective to training and coaching others.

Based in Bermuda, I enjoy rum-fuelled island life with my beautiful wife and son, while helping others to create their own success stories.

Harry Munro CEng MIMechE MSc BEng (Hons.)
Founder of the School of Simulation

If you would like to join my complimentary free masterclass “How to Become a Go-To Expert in Simulation with Python” you can register for access here:
https://simulation.teachem.digital/webinar-signup

School of Simulation