

SIMULATION IN PYTHON WITH SIMPY

A GENTLE INTRODUCTION TO THE WORLD OF
SIMPY AND ANALYSING SIMULATIONS WITH
REAL-WORLD EXAMPLES

HARRY MUNRO



Copyright © 2025 by Harry Munro

All rights reserved.

No part of this book may be reproduced in any form or by any electronic or mechanical means, including information storage and retrieval systems, without written permission from the author, except for the use of brief quotations in a book review.

TESTIMONIALS

I have worked extensively with simulation tools throughout my career. Harry's guide to SimPy provided clarity and practical insights beyond what I've encountered in official documentation. I believe this guide will be highly valuable to professionals and researchers alike.

GYÖRGY LIPOVSZKI, ASSOCIATE PROFESSOR
(RETD) OF SIMULATION, COMPUTER
CONTROLLED SYSTEMS AND SIGNAL
PROCESSING

I really enjoyed reading your guide as it provides really structured overview especially compared to SimPy's own documentation. Cannot wait to see more of your guides.

OGUZHAN INAL, INDUSTRIAL ENGINEER

*To my mother and father,
Who taught me patience in life's long queue,
And the strength to model all that's true.*

*To my wife,
The heart behind every simulation's beat,
Who brings balance to the chaos I meet.*

*To my son,
A spark of curiosity in every run,
May your journey be full of wonder, yet never done.*

*To my cat,
A master of randomness and serene delay,
In whose quiet steps, simulations play.*

... in real life mistakes are likely to be irrevocable. Computer simulation, however, makes it economically practical to make mistakes on purpose. If you are astute, therefore, you can learn much more than they cost. Furthermore, if you are at all discreet, no one but you need ever know you made a mistake.

JOHN H. MCLEOD WITH CO-AUTHOR JOHN
OSBORN, IN *NATURAL AUTOMATA AND
USEFUL SIMULATIONS* EDITED BY H. H. PATTEE
ET AL. (1966)

CONTENTS

Preface	x
Introduction	xii
1. WHAT IS DISCRETE-EVENT SIMULATION (DES)?	1
Key Characteristics of DES	2
Applications of DES	2
Advantages of DES	3
Limitations of DES	3
2. SIMPY'S ROLE IN DES	4
Why SimPy	4
A Personal Example: : Modelling Green Hydrogen for Decarbonising Mining	5
Applications in Academia and Industry	7
3. SIMPY BASICS	8
Installation	8
Core Concepts	8
Simulation Flow	13
Summary of Key Functions	13
4. WRITING A SIMPLE SIMPY PROGRAM	15
Step 1: Define the Environment	15
Step 2: Define the Process	15
Step 3: Add the Process to the Environment	16
Step 4: Run the Simulation	16
Complete Simple Example	17
Explanation of Output	17
What's Happening Under the Hood	18
Customising the Simulation	18
5. KEY COMPONENTS IN SIMPY	19
Events and Processes	19
Timeouts (Pausing Processes)	20
Resources	20
Scheduling Events	21
Combining Resources and Processes	22

6. SIMULATING A QUEUE SYSTEM	24
The Basics of Queuing Systems	24
Defining the Customer Process	24
Defining the Resource (Service Counter)	25
Adding Multiple Customers	26
Introducing Stochastic Delays Between Customer Arrivals	27
Understanding the Output	28
Visualising the Queue	29
Modelling Multiple Entities Competing for Resources with Traceability	37
7. TIPS FOR EFFICIENT SIMULATIONS	43
Write Modular Code	43
2. Use Meaningful Variable Names	44
3. Manage Time Properly	44
4. Control Simulation Length	45
5. Avoid Overloading Resources	45
6. Track Statistics and Performance	46
Debugging and Validation	46
Plan for Scalability	47
8. YOUR JOURNEY INTO SIMULATION	48
About the Author	51

PREFACE

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..!

Downloadable Code

Most of the code examples are available for download. You can access the code by downloading and unzipping the file at this link.

I hope you find this guide to simulation in SimPy helpful.

Yours sincerely,

Harry Munro CEng MIMechE MSc BEng (Hons.)

P.S. 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>

INTRODUCTION

Imagine orchestrating a symphony where every instrument strikes its note not in continuous harmony, but precisely at the right moment. The violins play as the curtains rise, the flutes chime at the crescendo, and the timpani echoes at just the right beat. This is more than music - it's precision, coordination, and design.

Discrete-Event Simulation (DES) embodies this concept in the world of systems and processes. It's a powerful tool that models complex systems as a sequence of distinct, impactful events. Whether optimising factory floors buzzing with machinery, streamlining patient care in bustling hospitals, or balancing data traffic across networks, DES allows you to model the pivotal moments that define system performance.

DES empowers you to ask and answer critical questions: What happens if a machine breaks down? How will patient waiting times change with more staff? Can this network handle a surge in traffic? Instead of guessing, DES provides precise, actionable insights based on realistic scenarios.

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. More-

over, 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 FOR

This 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.

A CASE STUDY: TRANSFORMING 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.

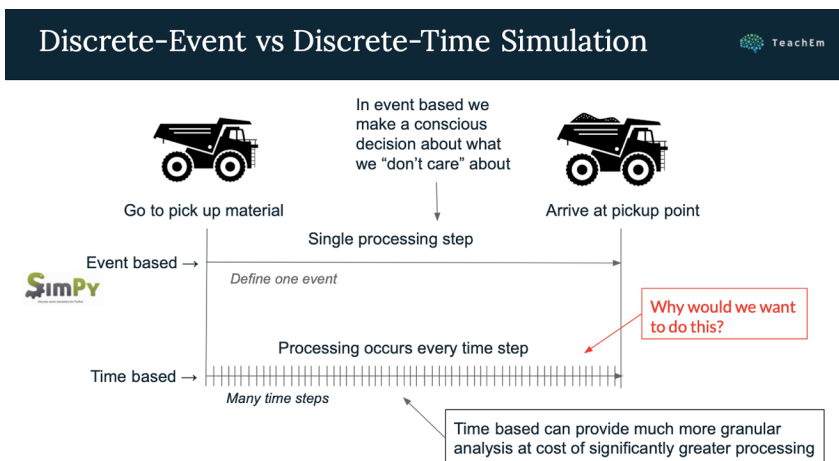
DOWNLOADABLE CODE

Most of the code examples in this book are available for download. You can access the code by downloading and unzipping the file at this link.

CHAPTER 1

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.



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.

CHAPTER 2

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.

WHY SIMPY

In the world of simulation, simplicity and flexibility are often at odds. Traditional simulation tools can be robust but intimidating, while simpler ones may lack the depth needed for complex systems. SimPy strikes a perfect balance by offering:

- **Straightforward Syntax:** Built on Python, SimPy is approachable for anyone familiar with Python programming. This simplicity allows you to focus on modelling logic rather than wrestling with complex configurations.

- **Versatility:** From queuing systems to manufacturing processes, SimPy can model a wide range of scenarios, making it suitable for both academic studies and real-world industry problems.
- **Efficiency:** By adopting a process-based approach, SimPy simplifies the creation and scheduling of events. It eliminates unnecessary computational overhead by focusing only on moments of change, ensuring that even large simulations run efficiently.

How SimPy Works

At its essence, SimPy revolves around three core components:

- **Processes:** Represent the dynamic entities in your system, such as customers queuing at a checkout or machines operating in a factory. Processes are defined using Python's generator functions, allowing them to pause and resume seamlessly.
- **Resources:** SimPy models shared assets like servers, workers, or machines that processes compete for. This is invaluable for simulating real-world systems where resource constraints significantly impact performance.
- **Events:** The building blocks of a simulation, representing key moments such as task completions or resource availability. SimPy efficiently schedules and manages these events to keep your system running smoothly.

A PERSONAL EXAMPLE: : MODELLING GREEN HYDROGEN FOR DECARBONISING MINING

One of the most impactful uses of SimPy in my career was modelling a **green hydrogen production system** for the mining industry.

. . .

The goal was to design a facility capable of producing and storing hydrogen to fuel hydrogen powered mining trucks, significantly reducing carbon emissions. With project costs running into the tens to hundreds of millions of dollars, it was crucial to optimise the design and estimate operational costs as early as possible in the project lifecycle.

Using SimPy, I created a simulation that modelled the entire hydrogen production and utilisation process:

- **Hydrogen Electrolysers:** Simulated production rates, downtimes, and energy consumption. Factoring in fluctuating renewable energy scenarios.
- **Storage Systems:** Modelled the dynamics of hydrogen storage, including filling, depletion, and buffer requirements to meet operational demands.
- **Demand Profiles:** Factored in fluctuating requirements for hydrogen-powered trucks and equipment across a typical mining operation.

The simulation revealed several key insights. For instance, it identified bottlenecks in storage capacity during peak demand periods and quantified the cost impact of different electrolyser configurations. This allowed the company to make informed decisions about scaling production capacity and selecting optimal technologies.

The result? The company avoided costly over-designs while ensuring the system met operational demands. The simulation also clarified the lifecycle costs, enabling better budget allocation and improving project feasibility assessments.

APPLICATIONS IN ACADEMIA AND INDUSTRY

SimPy's elegance and flexibility make it a favourite for both educational and industrial use:

- **In Academia:** SimPy is an excellent teaching tool, providing students with hands-on experience in building simulations. Its clear syntax makes it ideal for demonstrating the principles of DES.
- **In Industry:** Whether optimising supply chains, modelling hospital patient flows, or balancing data traffic in IT networks, SimPy provides the functionality needed to simulate and refine systems across various sectors. In high-stakes industries like mining or energy, its ability to evaluate multi-million-pound decisions is unparalleled.

SimPy isn't just a tool; it's a bridge between theory and practice. By combining Python's accessibility with DES's precision, it empowers users to model, test, and improve systems with confidence. As you progress through this guide, you'll see how SimPy transforms abstract concepts into practical solutions, ready to tackle the challenges of designing, evaluating and optimising modern systems.

CHAPTER 3

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.

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()
```

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}")
```

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}")
```

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_re-
quest(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 unit-less 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.

CHAPTER 4

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
Car parks at 14
```

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).

CHAPTER 5

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.

CHAPTER 6

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()
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))
env.run(until=20)

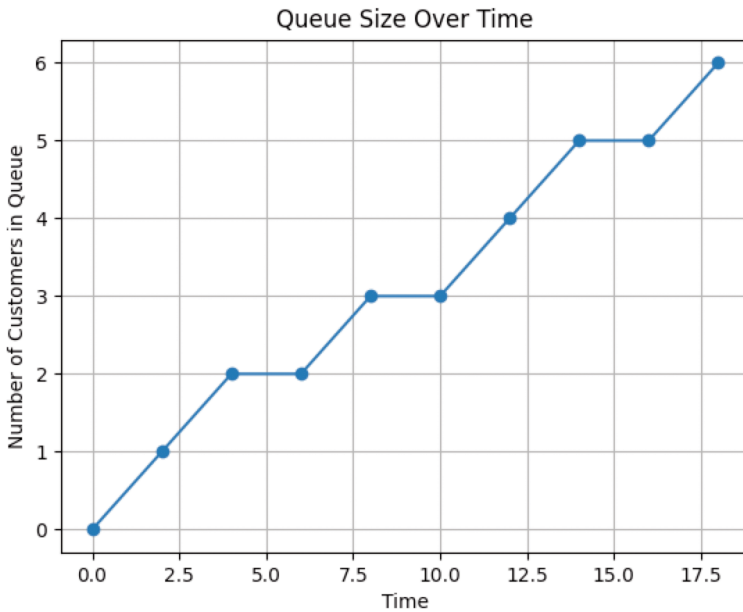
# Extract the time points and queue lengths
for plotting
    times, queue_sizes = zip(*queue_lengths)

```

```

# 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')

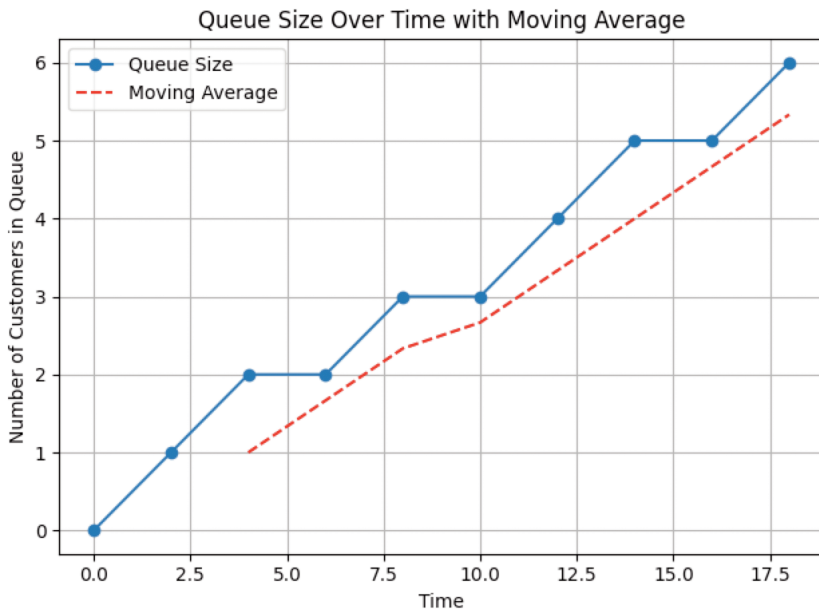
# Adjust the x-axis times to align the moving
average with the correct starting point
adjusted_times = times[window_size-1:] #
Times for the moving average plot
plt.plot(adjusted_times, moving_avg,
```

```

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()

```

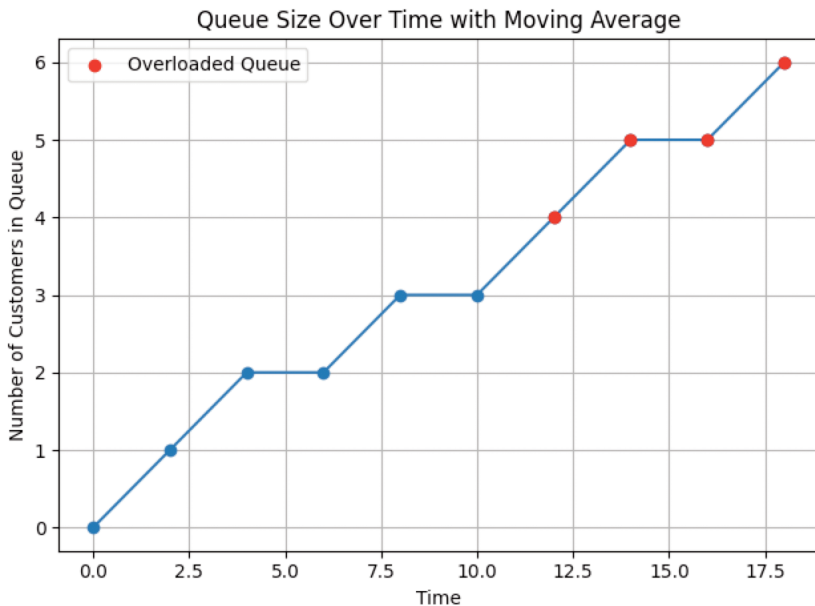


2. Colour-Coding Queue Overload:

- 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)
```

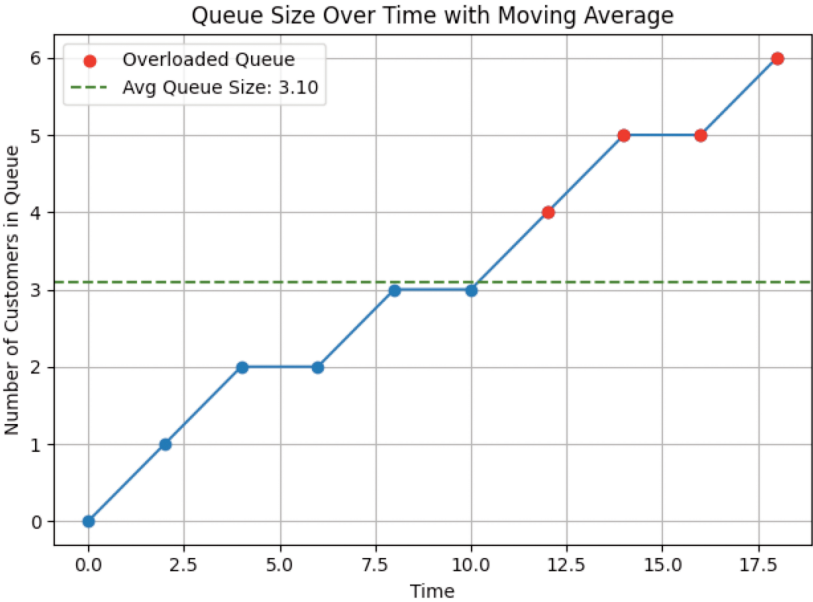
**3. Tracking Average Queue Size:**

- 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}')
```



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 2, meaning up to two 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 2
(e.g., 3 service desks)
counter = simpy.Resource(env, capacity=2)

# 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')

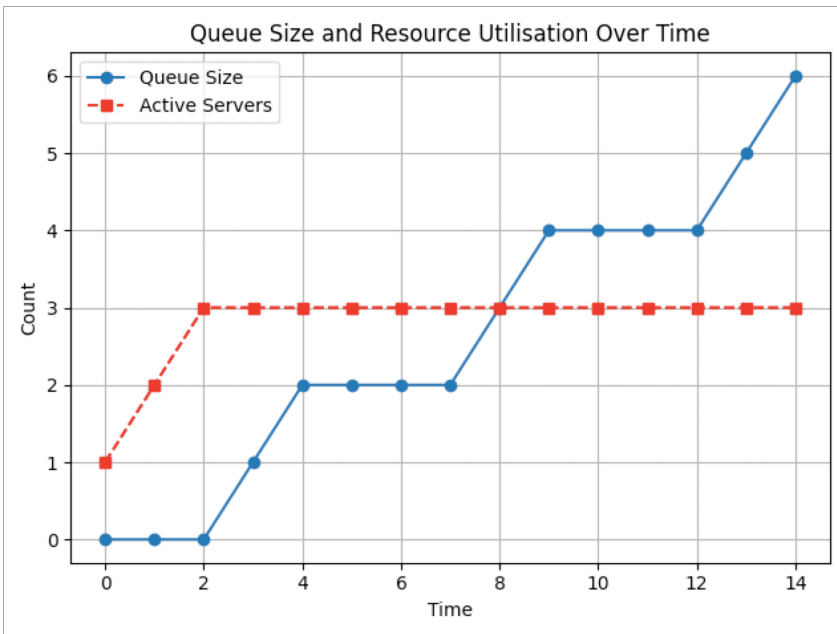
# Plot the number of active servers (resource
utilisation) over time
plt.plot(times_servers,      server_counts,
marker='s',  linestyle='--',  color='red',
label='Active Servers')

```

```

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()

```



Explanation of the Code

- **Resource with Capacity 3:**
 - `counter = simply.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.

CHAPTER 7

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.

2. 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:

```

def customer(env, counter):
    ...

```

3. 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.

4. 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
```

5. 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
```

6. 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.

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
```

Use your debugger. Interactive development environments will come with a debugging functionality which will allow you to step into your code and pause it at various times. Get familiar with the debugger in your development environment, it will make diagnosing problems with your simulation that much easier.

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.

CHAPTER 8

YOUR JOURNEY INTO SIMULATION

As we come to the end of this guide, it's clear that discrete-event simulation (DES) offers immense power in understanding complex systems. Whether you're modelling customer queues, manufacturing processes, or network traffic, the ability to simulate and optimise your systems opens up new opportunities for efficiency, cost savings, and innovation.

By learning SimPy, you've not only equipped yourself with the technical know-how but also gained insights into how real-world systems behave under various conditions. This knowledge is incredibly valuable, whether you're in academia, industry, or simply curious about the inner workings of dynamic systems.

But this is only the beginning of your journey. There's so much more to explore in the world of simulation:

- **Refining and Scaling:** As your skills grow, you'll find new ways to refine your models, making them more complex and scalable. Whether that's through adding more advanced

statistical models, optimisation techniques, or integrating with machine learning, the possibilities are limitless.

- **Industry-Specific Applications:** With what you've learned here, you're ready to apply DES in your field of work. From healthcare to logistics, simulation helps drive better decisions by providing data-backed insights. You can explore real-world challenges in your sector and experiment with solutions in a risk-free virtual environment.
- **Collaboration and Innovation:** The beauty of simulations is that they bring together multiple disciplines—engineering, data science, management, and beyond. As you continue building your expertise, consider how you might collaborate with others to solve bigger, more complex problems.

Thank you for taking this journey with me. I hope this guide has been an insightful companion, offering both a practical understanding of SimPy and a broader appreciation for the power of simulations. Keep experimenting, keep questioning, and most importantly, keep learning.

If you're ready to take your simulation skills to the next level, you can sign up to my free masterclass here: <https://simulation.teachem.digital/webinar-signup>

I wish you the best of luck in your simulation adventures!

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>

 [linkedin.com/in/harrymunro](https://www.linkedin.com/in/harrymunro)