SimPy Basics:

A Guidance Document On How To Use Python's Simulation Library

Group Number: 54
Authors: Kristen Hart and Kelsey Ritchey

Table of Contents

Abstract

This is a guidance document for SimPy, which is a free, easy to use, and well documented Python Library to code simulations. This document reviews key aspects of SimPy, alongside two example SimPy projects. The first example project is the Rowing Regatta simulation, authored by Kelsey Ritchey. The second example is the Dragon's Lair, authored by Kristen Hart. The goal of these examples is to showcase real applications of key SimPy functions, and display how one could apply the functions in their code. One major drawback to SimPy is that it cannot visualize the simulation as easily as some simulation software like Arena.

Background and Purpose

There are many different methods to simulate processes. This guidance document provides in-depth guidance on what SimPy is, how it can be used to build simulations, and our opinions on it. To better explain how to

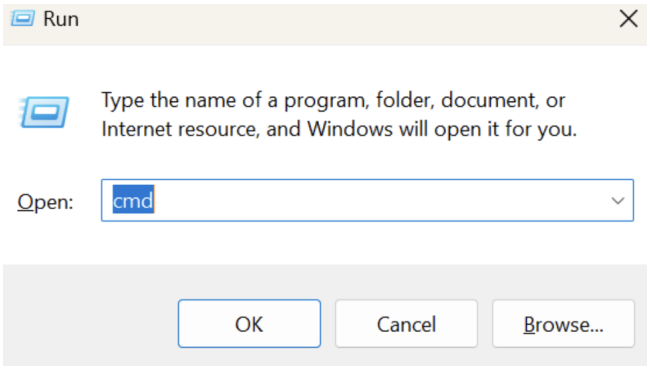
use SimPy, two example simulations have been coded (Dragon's Lair and Rowing Regatta) and are referred to throughout the document.

What is SimPy?

SimPy is a discrete event simulation library in Python that allows for processes to be modeled and run in a real time environment. It is a powerful tool that can be utilized across many different fields such as manufacturing, supply chain management, transportation and logistics, network simulation, and more, allowing for people to gain a strong understanding of the way that processes and resources interact with each other. This further leads to allowing for SimPy to be a useful tool for understanding how to optimize specific processes and events that occur in real time.

Installing SimPy

On Windows, SimPy can be installed through the Command Prompt or Powershell. You can navigate to the Command Prompt by pressing Windows+R, typing in "cmd", and hitting enter, or "OK". Once the Command Prompt is open, you just need to type "pip install simpy" to install the latest version of SimPy on their machine. In order to run pip commands, the user needs to ensure they have installed Python from the official Python webpage.



The process for installing on macOS and Linux distributions is similar. First, ensure that Python is installed on your machine, then open a Terminal, and run "pip install simpy".

```
C:\Users\krist>pip install simpy
Collecting simpy
  Downloading simpy-4.1.1-py3-none-any.whl.metadata (6.1 kB)
Downloading simpy-4.1.1-py3-none-any.whl (27 kB)
Installing collected packages: simpy
Successfully installed simpy-4.1.1
```

Environments

The SimPy Environment is a central component to SimPy that manages the simulation time and the events that occur within.

```
# Initialize the SimPy environment
env = simpy.Environment()
```

Time

Time in SimPy operates similarly to real life. SimPy processes events one after another, it does not process events at the same time. In the event that two events are scheduled to occur at the same time, SimPy will process the event that was scheduled first to occur first, following a first-in-first-out principle. It is important to note that this mimics how most events occur in real life. For example, consider the stock market's closure time at 4pm. It appears that all trading activity would stop at exactly 04:00:00.000. However, Trader A's final order may have been processed at 04:00:00.123 PM while Trader B's final order was processed at 04:00:00.456 PM. Although on the surface level, both trader's orders were processed at 4 PM, there was a slight time difference between them, and the two events did not occur at the exact same time. This is how SimPy treats events within a simulation.

In SimPy, Python generator functions, or "processes" in SimPy, are used to model the behavior of the elements in the simulation over a period of time. Processes in SimPy can lead to events, which can trigger the

simulation to pause (i.e. a timeout). When a process is first created, it is considered to be a passive object. Simply put, this means that the process does not have any scheduled events.

To start or stop a SimPy process object, we can use the functions `activate` or `start`.

To turn a non-passive object into a passive object, you can use the `passivate` function. This can be quickly reversed by utilizing `reactivate`. To cancel an object and all its future events, you can use `cancel`. Note that trying to cancel a non-active process will cause an error, as cancel only works on active objects.

Another powerful SimPy function is `interrupt` which allows you to interrupt an active process object. Another process or external event can cause a process object to be interrupted. This allows for any external events or conditions to be handled before resuming the process. For example, consider how in the Dragon's Den simulation, the adventurers have a chance to interrupt the dragon's sleep cycle, adding more risk to the event.

Next, consider the important feature, `yield`. For example, in the Dragon's Lair simulation, yield is used to simulate the dragon's sleep and wake cycles. The dragon sleeps for 14 hours per day. The code below pauses the process of the dragon's sleep duration, which simulates the passage of time. Meanwhile, the loop in `dragon's_lair` continues to run, which simulates the behavior of the adventurers attempting to steal from the dragon's lair.

```
        # Dragon sleeps for 14 hours per day
        dragon_sleep = not dragon_sleep
        sleep_duration = 14 if dragon_sleep else 10
        yield env.timeout(sleep_duration)
```

So, `yield` is used within a process function to pause the execution of an event and schedule the process to resume at a later time, as shown in the dragon's lair.

Generators, Functions, and Events

Once the initial variables have been defined, a python class can be used to contain the simulation's generator functions. Before diving into what generators are and how they can be developed, you must first understand how to initialize the objects within the class. For the Rowing Regatta example, we used the `__init__` method:

```
#defining processes
class RowingRegatta:
    def __init__(self, env, boats_per_race):
        #initialize environment
        self.env = env
        #intialize number of boats per race
        self.boats_per_race = boats_per_race
        #boat count starts at 0
        self.boat_count = 0
        #creating a list of true/false to determine if the race is experienced or novice
        self.is_exp = [True] * nov_races + [False] * (total_boats // boats_per_race -
    nov_races)
        random.shuffle(self.is_exp)
        #start the run process everytime an instance is created
        self.action = env.process(self.run_regatta())
        #will record start times of races to compute the average race center
        self.race_start_times = []
        #will record the race type of experienced or novice
        self.race_types = []
```

Within the `__init__`, we initialize a number of objects with the most critical being the environment (`self.env`) and the run process (`self.action`). Both of these are applicable to any simulation. The other initialized objects are more specific to this particular example simulation:

- Number of boats per race (`self.boats_per_race`) was one of the initial variables assigned (see Appendix A for the full example).
- Number of boats that have completed the regatta (`self.boat_count`) starts at 0 when the simulation begins.
- A True/False list (`self.is_exp`) is created to determine if a specific race has experienced or novice (inexperienced) rowers.
- Race start times (`self.race_start_times`) and race types (`self.race_types`) can be leveraged to output information useful in post-simulation analysis (see the Analysis of Results section).

Once `__init__` is complete, you can begin developing generators. Generators are functions that outline the processes taking place within a simulation and create a simulation's events. They must contain a yield expression as this is what makes the simulation wait for a certain event to occur before proceeding. Using the same Rowing Regatta example as before, here is the first generator function:

```
    def boat(self, bow_num, start_flag):
        #boat arrives event
        arrival_time = random.expovariate(lambda_arrive)
        yield self.env.timeout(arrival_time)
        print(f'Bow Number {bow_num} arrived at time {self.env.now}')
        #is boat experienced or novice?
        is_exp = self.is_exp[(bow_num - 1) % self.boats_per_race]
        #boat gets locked on to stake boat event
        if is_exp:
            lockon_time = random.uniform(lower_lock_exp, upper_lock_exp)
        else:
            lockon_time = random.uniform(lower_lock_nov, upper_lock_nov)
        yield self.env.timeout(lockon_time)
        print(f'Bow Number {bow_num} locked on to stake boat at time {self.env.now}')
        #all boats waits for start flag event
        yield start_flag
        #boat races event
        race_time = random.gauss(mu_race, sigma_race)
        yield self.env.timeout(race_time)
```

Here is another example from the Dragon's Lair:

```
import simpy
import random

# Define the adventurers
class Adventurer:
    def __init__(self, name, luck, stealth, speed):
        self.name = name
        self.luck = luck
        self.stealth = stealth
        self.speed = speed
        self.alive = True # Each starts alive
        self.loots = {'diamonds': 0, 'rubies': 0, 'sapphires': 0, 'emeralds': 0, 'gold': 0} #
    Store the loot obtained from the lair in a dictionary
        self.successful_loots = 0

    def roll_dice(self):
        base_roll = random.randint(1, 20) # Roll 20 sided dice
        modifiers = self.luck + self.stealth + self.speed
        final_roll = base_roll if base_roll == 20 else base_roll + modifiers
        return base_roll, modifiers, final_roll # Final roll = dice roll + sum of each
    adventurer's modifiers

    def loot(self): # Randomize the loot obtained from each successful raid of the lair
        self.loots['diamonds'] += random.randint(5, 10)
        self.loots['rubies'] += random.randint(3, 7)
        self.loots['sapphires'] += random.randint(6, 8)
        self.loots['emeralds'] += random.randint(9, 14)
        self.loots['gold'] += random.randint(1,5)
        self.successful_loots += 1
```

Within each generator there can be any number of events. The Rowing Regatta example generator generates boats and contains the following:

- Timeout Events: Timeout events are triggered using yield `self.env.timeout()`. A process is yielded until an event takes place and once that event has occurred, the process resumes. These events allow time to pass within the simulation.
  - Boat Arrival At the Start Line Event: Boats have a random interarrival time following an exponential distribution where lambda is .5 minutes.
  - Boats Locking On To Stakeboat Event: This timeout event is slightly more complicated than the first because we use the `is_exp` attribute as a condition to determine how quickly the boats lock on to their stake boats. Experienced boats lock on with a Uniform distribution between 1-3 minutes and novice boats with a Uniform distribution between 1-5 minutes (i.e. it sometimes takes novice boats longer to do this).
    - Brief Rowing Lesson: At the start line of many rowing regattas, there are floating platforms called stakeboats where a volunteer lays on their stomach off the edge of the platform. Boats must maneuver themselves in their assigned lane for the race so the stern (back) of their boat can be caught and held by the volunteer. This is called 'locking on to the stake boat'. Volunteers are then asked by the race officials to pull their boat in towards them or push it away from them until the race officials can see that the bows (fronts) of all boats in the race are even. Then, the race can be started.

- Boat Races Event: Each boat's race time to complete the 2000 meter race course follows a Normal distribution with $\mu = 7.5$ minutes and $\sigma = 1.5$ minutes.

- Print Statements: These are useful for outputting what time an event occurs as the simulation runs. For the boat arrival event, it prints the arriving boat's bow number (unique identifier of each boat) and arrival time within the simulation (self.env.now). Self.env.now is a commonly used timestamp within SimPy simulations since it is the exact current time in the simulation. Here is a portion of that output:

```
Bow Number 2 arrived at time 0.26992128238903124
Bow Number 4 arrived at time 0.6900617684465471
Bow Number 5 arrived at time 1.1264762372523467
Bow Number 6 arrived at time 1.296054671364965
```

The second generator function, `run_regatta`, in the Rowing Regatta example contains a different type of event other than a Timeout event.

Shared Events: This event is different because it will occur at the same time for multiple entities (like boats) in the simulation. In this example, the shared event is the race starting. Triggering a shared event can be done by defining the event (`start_flag`), yielding the process for no time (`yield self.env.timeout(0)`), having the shared event succeed (`start_flag.succeed()`), and yielding the simulation until all boats in the race have finished (`yield self.env.all_of(boats)`) as seen in this example below of the `start_flag event`. Here is the code specific to this shared event (see Appendix A for the full code of the second generator):

```
            #common event for all boats to start race at same time
            start_flag = self.env.event()

```

```
            #pauses the process for 0 minutes so the start race event can be triggered
            yield self.env.timeout(0)
            #start race event triggered
            start_flag.succeed()
            #waiting for all boats to finish race event
            yield self.env.all_of(boats)
            print(f'{race_type} Race finished at time {self.env.now}')

```

- This example only triggers the `stat_flag` event successfully, but events can also be triggered to fail using `event_name.fail()`. For example, we could make the `start_flag` event fail if a certain condition is met such as a weather event such as a storm occurring that causes a race to be canceled.

Shared Resources

SimPy shared resources all hold something with a limited capacity and processes can put more in or take something out. Processes can't do this when there is no capacity available (it's full) or there is nothing left to take out (it's empty) and must wait in a queue. These resources are called via resource in Python. There are specialized versions of this which include processes in the queue being ordered by priority (`PriorityResource`) and processes that are designated important enough to take over a resource slot before another process has finished using it (`PreemptiveResource`). Here are short examples of both:

```
# PriorityResource
def priority_resource_ex(env, resource, priority):
    with resource.request(priority=priority) as req:
        yield req
        print(f'PriorityResource acquired at {env.now} with priority {priority}')
        yield env.timeout(1)
        print(f'PriorityResource released at {env.now}')
```

```
# PreemptiveResource
def preemptive_resource_ex(env, resource, preempt=False):
    with resource.request(preempt=preempt) as req:
        try:
            yield req
            print(f'PreemptiveResource acquired at {env.now}')
            yield env.timeout(1)
            print(f'PreemptiveResource released at {env.now}')
        except simpy.Interrupt:
            print(f'PreemptiveResource preempted at {env.now}')

```

Resources can be requested and released by a process within a generator function using `resource.request()` and `resource.release(request)`.

There are 3 main types of shared resources:

1. Resource: This is the most basic form of a resource. These resources have a limited capacity for the number of processes that can use it at one time and when there are no free slots available, processes will wait in a queue until a slot becomes available by another process completing their use of the resource.

```
# Resource
def resource_ex(env, resource):
    with resource.request() as req:
        yield req
        print(f'Resource acquired at {env.now}')
        yield env.timeout(1)
        print(f'Resource released at {env.now}')

```

2. Store: This resource type allows processes to put Python objects in the resource and take them out. These Python objects do not need to be homogeneous (e.g. different types of cars at a car dealership).

```
# Store
def store_ex(env, store):
    yield store.put('item')
    print(f'Item put in store at {env.now}')
    item = yield store.get()
    print(f'Item retrieved from store at {env.now}')

```

3. Container: These store a continuous (e.g. coffee) or discrete (e.g. race medals) amount of a homogeneous something that similar to Store resources, allows processes to put more in and take more out, but if a process tries to take more out than the amount contained, it has to wait until more is put in.

```
# Container
def container_ex(env, container):
    yield container.put(5)
    print(f'Container now has {container.level} at {env.now}')
    yield container.get(3)
    print(f'Container now has {container.level} at {env.now}')

```

Creating and Running the Simulation

Now that the simulation has been created, it can be run. This is done via env.run(). You can either specify an end time, or run until all events are processed. In the Dragon's Lair example, the simulation is run until one adventurer remains alive, or until all have been killed by the dragon.

```
# Initialize the SimPy environment
env = simpy.Environment()

# Run the dragon's lair and simulation
env.process(dragon_lair(env, dragon_sleep, adventurers))
env.run()

```

Analysis of Results

Once a simulation has been run, analysis of the results can be helpful to check the simulation validity, answer key questions, solve a specific problem, and more. Therefore, when coding a simulation in SimPy, it is helpful to incorporate code to help you track key data points within your processes that can be used to analyze the simulation. Using the Rowing Regatta again as an example, let's say we built this simulation before a regatta to determine the average expected race centers (times between the start of each race), so we can ensure the race schedule we want to use can accommodate the expected race centers. Here are the relevant code snippets from the second generator function (see Appendix A for the full code of the second generator):

```
    #recording when race starts
    race_start_time = self.env.now

```

```
    #calculating average race center
    if len(self.race_start_times) > 1:
            race_centers = [
                self.race_start_times[i] - self.race_start_times[i - 1]
                for i in range(1, len(self.race_start_times))]
            average_center = sum(race_centers) / len(race_centers)
            print(f'Average race center: {average_center:.2f} minutes')

```

For this example, we record when each race starts (`race_start_time`). Then, we calculate the race center for each race once more than 1 race has started. Lastly, we compute the average race center (`average_center`) during the simulated regatta, which is 15.43 minutes.

Final Thoughts

Overall, SimPy is a fantastic introduction into the simulation world. To get started, you need minimal Python knowledge. After a few days of learning Python, SimPy would be easy to introduce to new coders and anyone interested in learning how to perform and analyze simulations. However, to perform any complex and meaningful simulations and corresponding analyses, a higher level knowledge of Python will be required, which may be a barrier to some. Ultimately, this is hard to use as a downside against SimPy, as this comes from a lack of Python knowledge, rather than a lack of SimPy knowledge. There is a plethora of documentation of SimPy online that explains in depth how to use it and how it works. The easy and free access to detailed documentation is a huge boon for SimPy. Furthermore, Python is an extremely popular programming language, and therefore SimPy may be more interesting to learn and use for most. The biggest downside of SimPy is that it is harder to visualize results than other simulation software like Arena. Arena allows for the user to visually witness the on-going events, whereas in SimPy, the user is not actively watching the events unfold in a visual manner. That being said, due to Python's popularity and the fact that it is free, makes SimPy overall a fantastic, easy to use, and easy to access simulation language.

Future Work

Looking into the future, this guidance document can be built out to include even more information on how to better visualize simulations in SimPy. More detailed examples and explanations can be supplied for each of SimPy's functions, and more in-depth examples on how to use and apply SimPy can be made.

The Rowing Regatta example could also be enhanced to include other processes (e.g. boats using the limited dock space to get to the start line or get off the water once their race is complete). Identifying additional data points for the results analysis would also be valuable to evaluate how well the regatta is being run and where improvements can be made.

The Dragon's Lair example could potentially be the framework for a fantasy-style video game. Adding in more complex functions to actually stealing from the dragon, a user interface to enhance user interaction, and more, could help flesh out this SimPy example into a game.

Appendix A

```
#installing
!pip install simpy

#importing
import random
import simpy

#setting random seed for reproducibility
seed = 123

#total boats that plan to race
total_boats = 204
#boat interarrival distribution parameters. Exponential with a mean of 2 (1/2 = .5)
lambda_arrive = .5
#stake boat lock on distribution parameters for experienced rowers
lower_lock_exp = 1
upper_lock_exp = 3
#stake boat lock on distribution parameters for novice rowers
lower_lock_nov = 1
upper_lock_nov = 5
```

```
#race distribution parameters
mu_race = 7.5
sigma_race = 1.25
#number of novice races in the regatta
nov_races = 12

#defining processes
class RowingRegatta:
    def __init__(self, env, boats_per_race):
        #initialize environment
        self.env = env
        #intialize number of boats per race
        self.boats_per_race = boats_per_race
        #boat count starts at 0
        self.boat_count = 0
        #creating a list of true/false to determine if the race is experienced or novice
        self.is_exp = [True] * nov_races + [False] * (total_boats // boats_per_race -
    nov_races)
        random.shuffle(self.is_exp)
        #start the run process everytime an instance is created
        self.action = env.process(self.run_regatta())
        #will record start times of races to compute the average race center
        self.race_start_times = []
        #will record the race type of experienced or novice
        self.race_types = []

    def boat(self, bow_num, start_flag):
        #boat arrives event
        arrival_time = random.expovariate(lambda_arrive)
        yield self.env.timeout(arrival_time)
        print(f'Bow Number {bow_num} arrived at time {self.env.now}')
        #is boat experienced or novice?
        is_exp = self.is_exp[(bow_num - 1) % self.boats_per_race]
        #boat gets locked on to stake boat event
        if is_exp:
            lockon_time = random.uniform(lower_lock_exp, upper_lock_exp)
        else:
            lockon_time = random.uniform(lower_lock_nov, upper_lock_nov)
        yield self.env.timeout(lockon_time)
        print(f'Bow Number {bow_num} locked on to stake boat at time {self.env.now}')
        #all boats waits for start flag event
        yield start_flag
        #boat races event
        race_time = random.gauss(mu_race, sigma_race)
        yield self.env.timeout(race_time)

    def run_regatta(self):
        #checking that we still have boats that haven't raced yet
        while self.boat_count < total_boats:
            #recording boats in a race
            boats = []
            #common event for all boats to start race at same time
            start_flag = self.env.event()
            #getting the maximum number of boats in per race
            for _ in range(self.boats_per_race):
                if self.boat_count >= total_boats:
                    break
                self.boat_count += 1
                boats.append(env.process(self.boat(self.boat_count, start_flag)))
            #recording race type
            race_type = "Experienced" if any(self.is_exp[(self.boat_count - 1) % self.
    boats_per_race:
                                                    (self.boat_count - 1) % self.
    boats_per_race + 6]) else "Novice"
            self.race_types.append(race_type)
            #recording when race starts
            race_start_time = self.env.now
            self.race_start_times.append(race_start_time)
            #pauses the process for 0 minutes so the start race event can be triggered
            yield self.env.timeout(0)
            #start race event triggered
            start_flag.succeed()
            #waiting for all boats to finish race event
            yield self.env.all_of(boats)
            print(f'{race_type} Race finished at time {self.env.now}')

```

```
        #calculating average race center
        if len(self.race_start_times) > 1:
            race_centers = [
                self.race_start_times[i] - self.race_start_times[i - 1]
                for i in range(1, len(self.race_start_times))]
            average_center = sum(race_centers) / len(race_centers)
            print(f'Average race center: {average_center:.2f} minutes')

#set up and run sim
print('Rowing Regatta')
random.seed(seed)
#create environment and setup
env = simpy.Environment()
regatta = RowingRegatta(env, boats_per_race=6)
#run sim
env.run()

```

Appendix B

```
import simpy
import random

# Define the adventurers
class Adventurer:
    def __init__(self, name, luck, stealth, speed):
        self.name = name
        self.luck = luck
        self.stealth = stealth
        self.speed = speed
        self.alive = True # Each starts alive
        self.loots = {'diamonds': 0, 'rubies': 0, 'sapphires': 0, 'emeralds': 0, 'gold': 0} # Store the loot obtained from the lair in a dictionary
        self.successful_loots = 0

    def roll_dice(self):
        base_roll = random.randint(1, 20) # Roll 20 sided dice
        modifiers = self.luck + self.stealth + self.speed
        final_roll = base_roll if base_roll == 20 else base_roll + modifiers
        return base_roll, modifiers, final_roll # Final roll = dice roll + sum of each adventurer's modifiers

    def loot(self): # Randomize the loot obtained from each successful raid of the lair
        self.loots['diamonds'] += random.randint(5, 10)
        self.loots['rubies'] += random.randint(3, 7)
        self.loots['sapphires'] += random.randint(6, 8)
        self.loots['emeralds'] += random.randint(9, 14)
        self.loots['gold'] += random.randint(1,5)
        self.successful_loots += 1

# Stat modifiers for each adventurer
adventurers = [
    Adventurer('Sylas the Rogue', luck=0, stealth=3, speed=3),
    Adventurer('Lucian the Wizard', luck=3, stealth=1, speed=1),
    Adventurer('Belgrom the Dwarf', luck=1, stealth=1, speed=-2),
    Adventurer('Keldan the Elf', luck=2, stealth=2, speed=2)
]

# Define the Dragon's Lair environment
def dragon_lair(env, dragon_sleep, adventurers):
    while sum(adventurer.alive for adventurer in adventurers) > 1:
        print(f"Time {env.now}: The dragon is {'asleep' if dragon_sleep else 'awake'}")
        for adventurer in adventurers:
            if adventurer.alive: # Only living adventurers can try to steal from the lair
                base_roll, modifiers, final_roll = adventurer.roll_dice()
                print(f"{adventurer.name} rolls a {base_roll} + {modifiers} = {final_roll}")

                if final_roll >= 20:
                    print(f"{adventurer.name} successfully steals from the lair without breaking a sweat. The dragon slumbers.")
                    adventurer.loot()
                elif final_roll > 15:
                    print(f"{adventurer.name} successfully steals from the lair!")
                    adventurer.loot()
```

```
                elif final_roll > 10:
                    print(f"{adventurer.name} stumbles...")
                    if not dragon_sleep:
                        print(f"...the dragon wakes up. {adventurer.name} was scoured by flames.")
                        adventurer.alive = False
                    else:
                        print(f"{adventurer.name} made it out! The dragon slumbers.")
                        adventurer.loot()
                else:
                    print(f"{adventurer.name} was scoured by flames.")
                    adventurer.alive = False

        # Dragon sleeps for 14 hours per day
        dragon_sleep = not dragon_sleep
        sleep_duration = 14 if dragon_sleep else 10
        yield env.timeout(sleep_duration)

    # Check to see if all adventurers died
    if sum(adventurer.alive for adventurer in adventurers) == 0:
        print("\nOh no! The whole party was scoured by flames. No loot was obtained from the dragon's lair.")
    else:
        # If they did not all die, see who is the last one standing
        last_adventurer = None
        for adventurer in adventurers:
            if adventurer.alive:
                last_adventurer = adventurer

        if last_adventurer:
            # Print final loot screen
            loot = last_adventurer.loots
            print("\nFinal Results:")
            print(f"{last_adventurer.name} was the last one standing.")
            print(f"The final loot is {loot['diamonds']} diamonds, {loot['rubies']} rubies, {loot['sapphires']} sapphires, {loot['emeralds']} emeralds, and {loot['gold']} pounds of gold. Shiny!")
            print(f"{last_adventurer.name} looted from the dragon's lair {last_adventurer.successful_loots} times before calling it")
    for adventurer in adventurers:
        if not adventurer.alive:
            print(f"{adventurer.name} had {adventurer.successful_loots} successful loot(s) before dying.")

# Dragon starts asleep
dragon_sleep = True

# Initialize the SimPy environment
env = simpy.Environment()

# Run the dragon's lair and simulation
env.process(dragon_lair(env, dragon_sleep, adventurers))
env.run()

```

Appendix C

```
import simpy

# Resource
def resource_ex(env, resource):
    with resource.request() as req:
        yield req
        print(f'Resource acquired at {env.now}')
        yield env.timeout(1)
        print(f'Resource released at {env.now}')

# PriorityResource
def priority_resource_ex(env, resource, priority):
    with resource.request(priority=priority) as req:
        yield req
        print(f'PriorityResource acquired at {env.now} with priority {priority}')
        yield env.timeout(1)
        print(f'PriorityResource released at {env.now}')
```

```python

# PreemptiveResource
def preemptive_resource_ex(env, resource, preempt=False):
    with resource.request(preempt=preempt) as req:
        try:
            yield req
            print(f'PreemptiveResource acquired at {env.now}')
            yield env.timeout(1)
            print(f'PreemptiveResource released at {env.now}')
        except simpy.Interrupt:
            print(f'PreemptiveResource preempted at {env.now}')

# Store
def store_ex(env, store):
    yield store.put('item')
    print(f'Item put in store at {env.now}')
    item = yield store.get()
    print(f'Item retrieved from store at {env.now}')

# Container
def container_ex(env, container):
    yield container.put(5)
    print(f'Container now has {container.level} at {env.now}')
    yield container.get(3)
    print(f'Container now has {container.level} at {env.now}')

env = simpy.Environment()

# Resource
resource = simpy.Resource(env, capacity=1)
env.process(resource_ex(env, resource))

# PriorityResource
priority_resource = simpy.PriorityResource(env, capacity=1)
env.process(priority_resource_ex(env, priority_resource, priority=1))
env.process(priority_resource_ex(env, priority_resource, priority=0))

# PreemptiveResource
preemptive_resource = simpy.PreemptiveResource(env, capacity=1)
p1 = env.process(preemptive_resource_ex(env, preemptive_resource, preempt=False))
env.process(preemptive_resource_ex(env, preemptive_resource, preempt=True))

# Store
store = simpy.Store(env, capacity=1)
env.process(store_ex(env, store))

# Container
container = simpy.Container(env, init=0, capacity=10)
env.process(container_ex(env, container))

env.run()

```

Works Cited

Python 3 Documentation, Python Software Foundation. The yield Expression. Accessed 11 July 2024. docs.python.org/3/reference/expressions.html#yieldexpr.

PythonHosted. SimPy Manual: Starting and Stopping SimPy Process Objects. Accessed 12 July 2024. pythonhosted.org/SimPy/Manuals/Manual.html#starting-and-stopping-simpy-process-objects.

SimPy Documentation. Basic Concepts. Accessed 27 June 2024. simpy.readthedocs.io/en/latest/simpy_intro/basic_concepts.html.

———. Events. Accessed 12 July 2024. simpy.readthedocs.io/en/latest/topical_guides/events.html.

———. Resources. Accessed 11 July 2024. simpy.readthedocs.io/en/latest/topical_guides/resources.html.

Wagner, Gerd. Intro to SimPy: Discrete Event Simulation in Python. 2014, Accessed 27 June 2024. YouTube, %20%5Curl%7Bwww.youtube.com/watch?v=Bk91DoAEcjY%7D.