Maroun Semaan Faculty of
Engineering and Architecture

Discrete Event Simulation

INDE 504

Introduction to SimPy

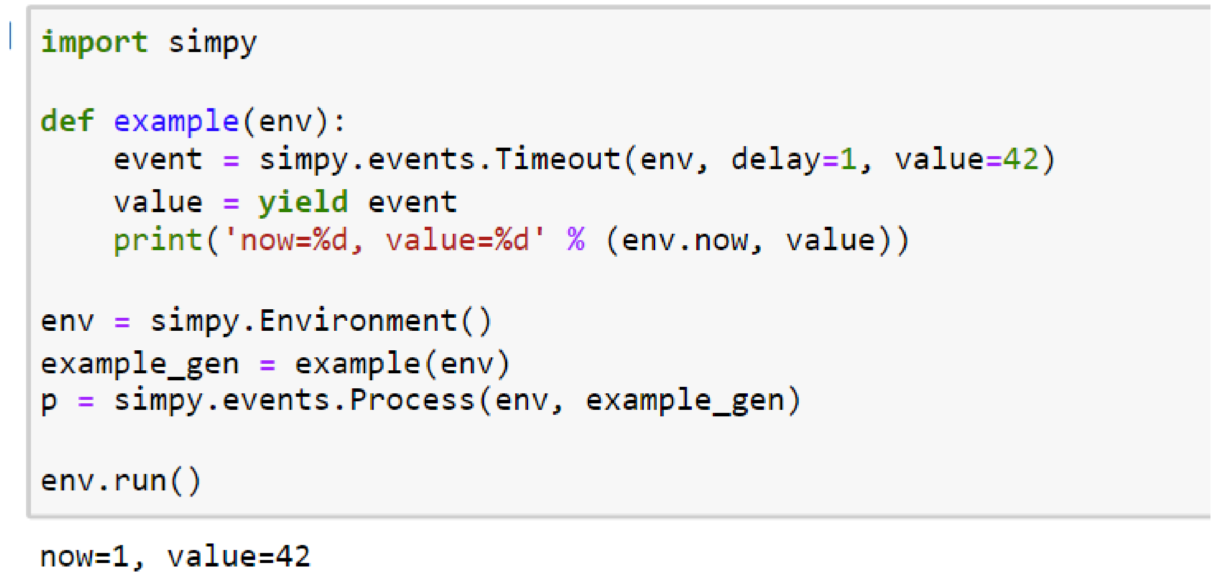
Overview

- **SymPi** is a process-based discrete-evet simulation framework based on standard Python. It seems to be the best available option as it is (i) process oriented which allows skipping the tedious task of coding the logic of individual events, (ii) well-documented, and (iii) widely demonstrated with several applications
- This will provide an open-source tool for process oriented simulation capable of competing with the high-cost commercial packages, such as Arena and AnyLogic, and can be easily integrated with other activities in the data lifecycle
- In SimPy, we define processes by generator functions and can be used to model active components like customers, vehicles or agents.

Basic Concepts

- We model the behavior of active components (like vehicles, customers or messages) with **processes.**
- These processes live in **environments.**
- Environments interact with each other via **events.**
- **Processes** are described by **generators.** During their lifetime, they create **events** and **yield** them in order to wait for them to happen.
- When a process yields an event, it gets **suspended** (it is waiting for the event to happen). In other words, it tells the simulation engine “I am waiting for this event to occur, so pause it here. SimPy **resumes** the process, when the event occurs. We say that the event is **processed.** Multiple processes can wait for the same event to happen. SimPy resumes them in the same order in which they yielded that event.
- An important event type is **Timeout.** Events of this type are processed after a certain amount of simulated time has passed. They allow a process to sleep (or hold its state) for the given time. A **Timeout** and all other events can be created by calling the appropriate method of the **Environment** that the process lives in. For example: **Environment.timeout()**

Basic Concepts - Simple Example



```
import simpy

def example(env):
    event = simpy.events.Timeout(env, delay=1, value=42)
    value = yield event
    print('now=%d, value=%d' % (env.now, value))

env = simpy.Environment()
example_gen = example(env)
p = simpy.events.Process(env, example_gen)

env.run()
```

```
now=1, value=42
```

1) Define “example” Function: we need to define a Python function called **“example”**, which represents a **process** in this simulation. It takes one argument, **“env”,** which is the simulation environment in which the process will run. It will be used to involve a **“Timeout” event.**

Basic Concepts - Simple Example

2) Create “Timeout” Event: inside the **“example” function,** we create a **“timeout” event** using the **“simpy.events.Timeout” constructor.** The **“env”** parameter specifies the simulation environment, **“delay”** is set to 1, indicating that this event will occur after a delay of 1 time unit in the simulation. **“Value”** is set to 42, which is the value that the event will produce when it occurs. The Timeout schedules itself at **now + delay,** and that’s why the environment argument is required.

3) Yield “Timeout” Event: we use the keyword **“yield”** to pause the execution of the “**example**” function and wait for the **“Timeout”** event to occur. When it occurs, the value 42 is assigned to the variable **“value”.**

4) Print Simulation Information: After the **“Timeout”** event occurs and the value is assigned to **“value”,** the current simulation time **“env.now”** and the value are printed to the console.

5) Create Simulation Environment: A SimPy simulation environment **“env=simpy.Envrionment”** is created. This environment is necessary for managing and running the simulation.

6) Create a Process: A SimPy process **“p”** is created using **“simpy.events.Process”.** This process is associated with the **“example”** function, and it represents the execution of the **“example”** function within the simulation environment.

7) Run the Simulation: Finally, the simulation is started by calling **“env.run()”.** This command causes the simulation environment to execute the **“example”** process. The simulation continues until there are no more events to process.

Environments

- **An environment** manages the simulation time as well as the scheduling and processing of events. It also provides means to step through or execute the simulation.
- **Simulation control:** in SimPy, you can run your simulation until there are no more events, until a certain simulation tine is reached, or until a certain event is triggered. Also, you can step through the simulation event by event, and you can mix all of these things together as you need.
- **Most important method: env.run().** If you run this command without any argument, it will run the simulation until thee are no more events left. You can also stop your simulation when a certain simulation time is reached by passing the desired time to **env.run()** using the **until** parameter. Example: **env.run(until=10).** The simulation will stop when the internal clock reaches 10 time units.

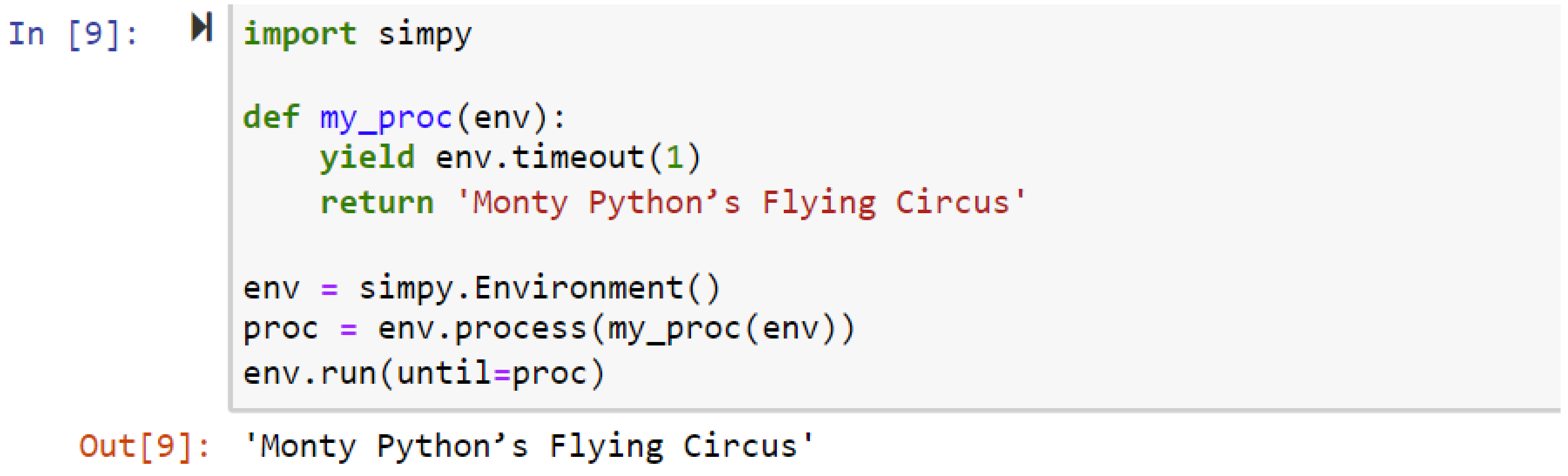
  Instead of passing a number to **env.run(),** you can also pass any event (example: **Timeout event)**, so will return when the event has been processed. Example: **env.run(until=env.timeout(5))** is equivalent to **env.run(until=5).**

  Remember also that a process is an event too!
- **State access: "environment.now"** allows you to get the current simulation time. The simulation time is increased via **Timeout events.** By default, **now** starts at 0,but you can pass an **initial_time** to the **Envrionment** to use something else, if needed.
- **Environment.active_process** will generate the currently active process of the environment, when called.

Environments

- **Shortcuts:** As previously mentioned, to create events, you should import **simpy.events,** instantiate the event class and pass a reference to the environment to it. However, the **Environment** provides some shortcuts for event creation. For example, **Environment.event()** is equivalent to **simpy.events.Event(env).**
- Other shortcuts are: **Environment.process()** and **Envrionment.timeout().**

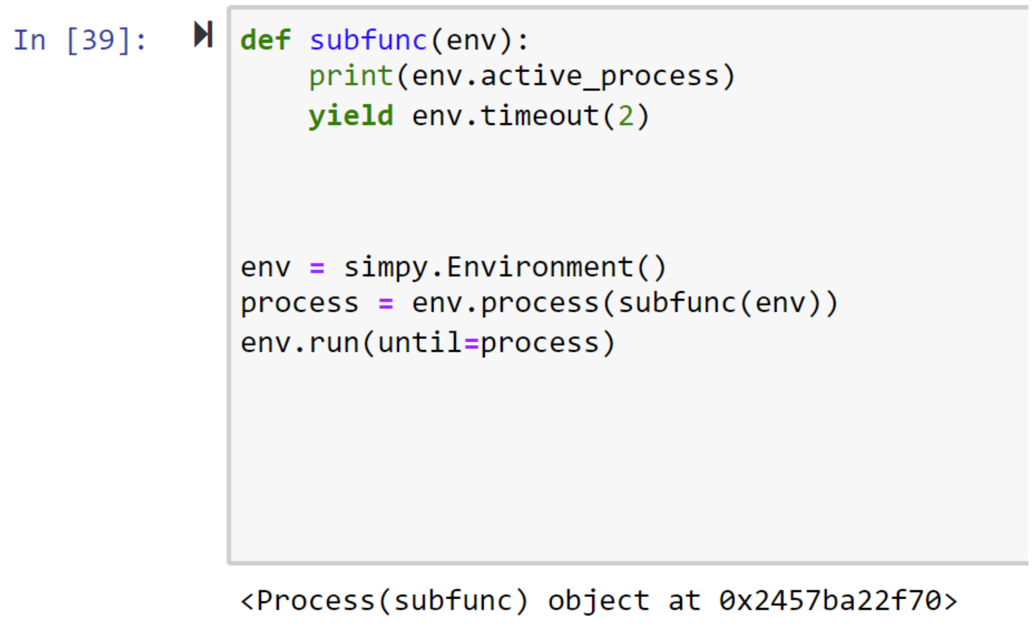
Environments – Simple Example 1



```
In [9]:  import simpy

          def my_proc(env):
              yield env.timeout(1)
              return 'Monty Python’s Flying Circus'

          env = simpy.Environment()
          proc = env.process(my_proc(env))
          env.run(until=proc)

Out[9]:   'Monty Python’s Flying Circus'
```

- This example defines a process **“my_proc”** that waits for 1 time unit and then returns a string. The simulation environment **“env”** is used to manage and run the simulation. The simulation is run until the **“my_proc”** process completes, but since the process does not have any further actions or events, the simulation effectively ends after waiting for 1 time unit.

Environments – Simple Example 2



```
In [39]:  def subfunc(env):
              print(env.active_process)
              yield env.timeout(2)



          env = simpy.Environment()
          process = env.process(subfunc(env))
          env.run(until=process)

          <Process(subfunc) object at 0x2457ba22f70>
```

- This example defines a process that prints the value of “**env.active.process**” (which is expected to be the **process** itself) and then yields for 2 time units. The simulation environment runs until this process is completed, allowing you to observe the active process within the simulation.

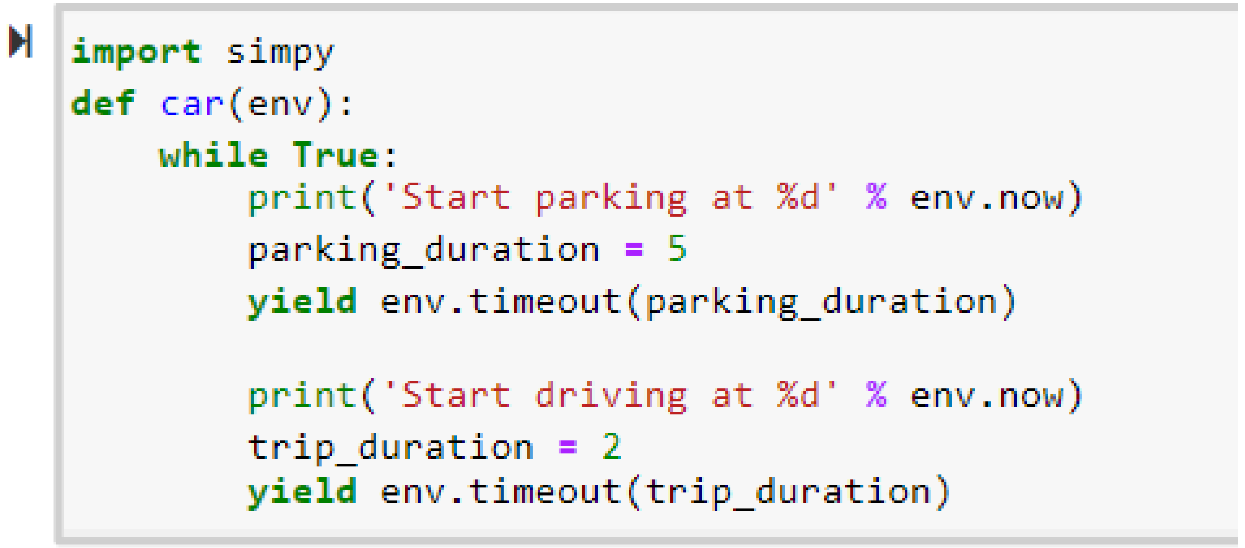
Events

- In SimPy, we have a variety of event types for various purposes.

```
events.Event
|
+— events.Timeout
|
+— events.Initialize
|
+— events.Process
|
+— events.Condition
|    |
|    +— events.AllOf
|    |
|    +— events.AnyOf
.
.
.
```

A First Process of an event in an Environment

A car will alternatively drive and park for a while. When it starts driving (or parking), it will print the current simulation time.



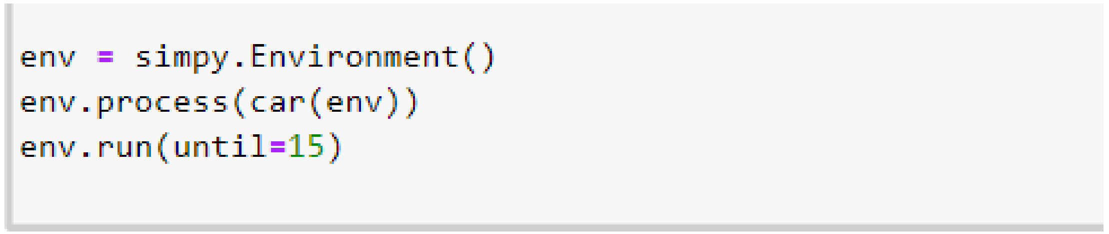
```
import simpy
def car(env):
    while True:
        print('Start parking at %d' % env.now)
        parking_duration = 5
        yield env.timeout(parking_duration)

        print('Start driving at %d' % env.now)
        trip_duration = 2
        yield env.timeout(trip_duration)
```

- We need to define a Python function called **"car"**, which represents a **process** in this simulation. It takes one argument, **"env"**, which is the simulation environment in which the process will run.

A First Process of an event in an Environment

- The **environment (env)** is required to create new events.
- The behavior of the process (the car) is described in an infinite loop (**while True).** The car process will continue running indefinitely, simulating the car's behavior repeatedly until the simulation is stopped manually.
- The **"parking_duration"** variable represents the amount of time that the car will spend parking.
- The **print** function is sued to display a message indicating that the car starts driving. **"env.now"** gives the current simulation time.
- **"yield"** is used in Python within a generator function (a process in our case) to temporarily pause the execution of the process and yield control back to the simulation environment. It yields an event called **"timeout"** with a duration of **"parking_duration"** (in this case, 5 time units). The simulates the car being parked for 5 time units.
- The same rationale applies for the trip duration while starting to drive.

A First Process of an event in an Environment



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

```
Start parking at 0
Start driving at 5
Start parking at 7
Start driving at 12
Start parking at 14
```

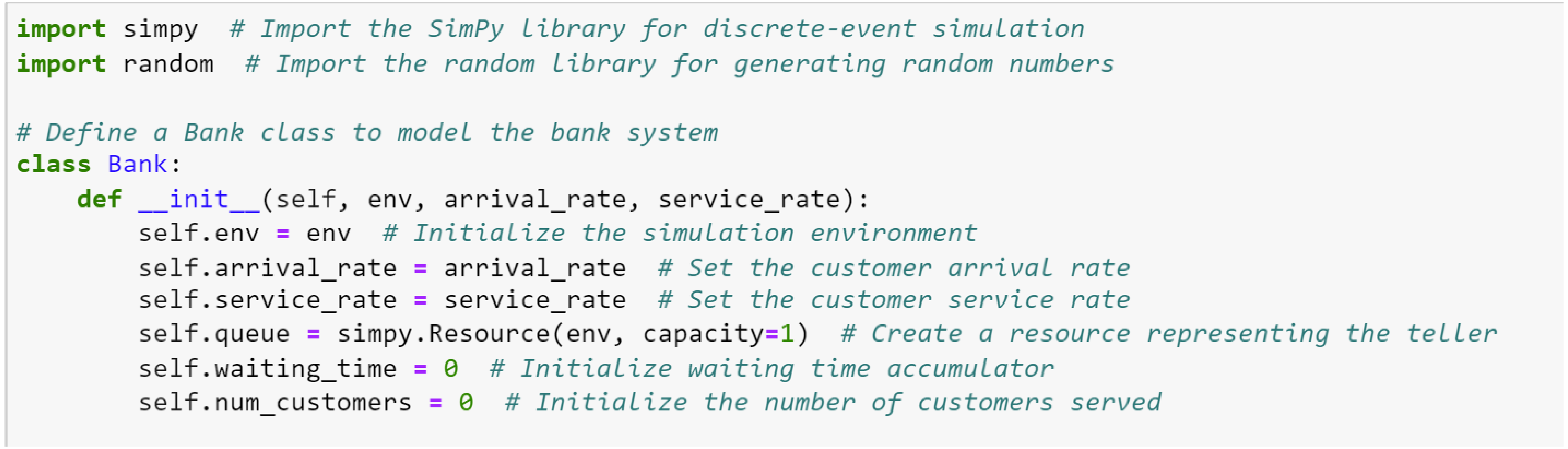
- Next, we need to create a SimPy simulation environment **“env=simpy.Environment()”**, which provides the infrastructure for managing and running the simulation.
- Then, we need to start the **“car” process** within our created simulation environment **“env”. “env.process(car(env))** associates the car process with the environment, allowing the simulation to manage its execution.
- Finally, we need to initiate the simulation and runs it until the specified time, in this case, until time 15. During this time, the simulation will execute the **“car” process**, update the simulation clock, and print messages to indicate when parking and driving events occur.

M/M/1 Queue Simulation

- Consider a simple M/M/1 queue at a single teller bank where customers arrive according to a Poisson Process at a rate of 9 customers per hour, and are served by the teller at the rate of 10 customers per hour, according to an exponential distribution.
- Simulate this queue.

M/M/1 Queue Simulation

1) Define the Bank Class

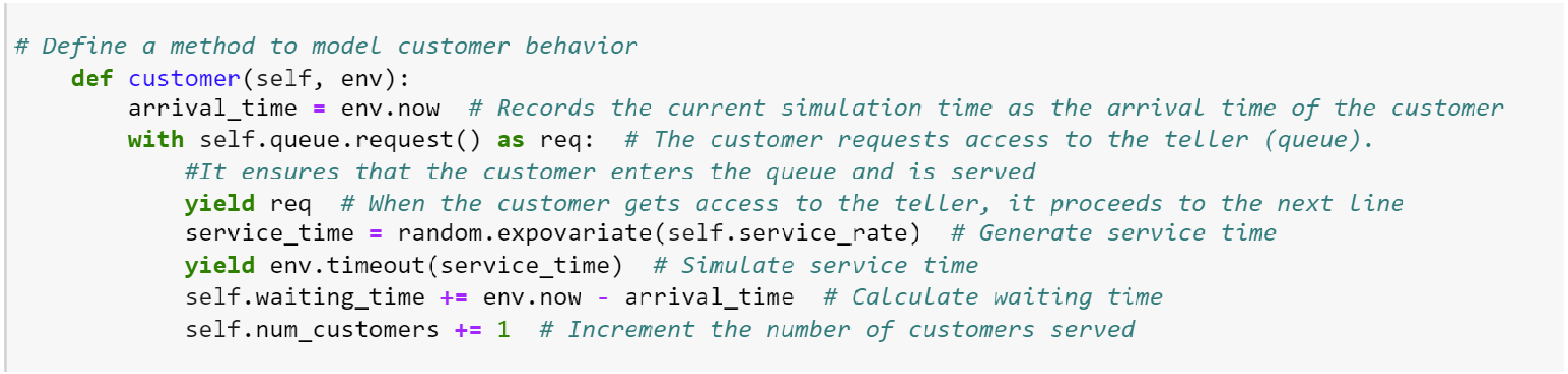


```python
import simpy  # Import the SimPy library for discrete-event simulation
import random  # Import the random library for generating random numbers

# Define a Bank class to model the bank system
class Bank:
    def __init__(self, env, arrival_rate, service_rate):
        self.env = env  # Initialize the simulation environment
        self.arrival_rate = arrival_rate  # Set the customer arrival rate
        self.service_rate = service_rate  # Set the customer service rate
        self.queue = simpy.Resource(env, capacity=1)  # Create a resource representing the teller
        self.waiting_time = 0  # Initialize waiting time accumulator
        self.num_customers = 0  # Initialize the number of customers served
```

Maroun Semaan Faculty of Engineering and Architecture

M/M/1 Queue Simulation

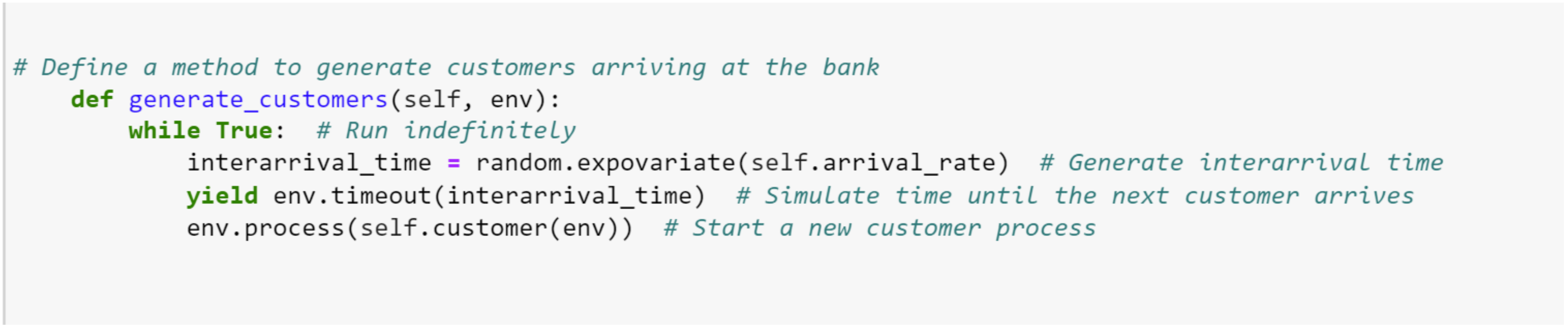
2) Model Customer Behavior



```
# Define a method to model customer behavior
    def customer(self, env):
        arrival_time = env.now  # Records the current simulation time as the arrival time of the customer
        with self.queue.request() as req:  # The customer requests access to the teller (queue).
            #It ensures that the customer enters the queue and is served
            yield req  # When the customer gets access to the teller, it proceeds to the next line
            service_time = random.expovariate(self.service_rate)  # Generate service time
            yield env.timeout(service_time)  # Simulate service time
            self.waiting_time += env.now - arrival_time  # Calculate waiting time
            self.num_customers += 1  # Increment the number of customers served
```

M/M/1 Queue Simulation

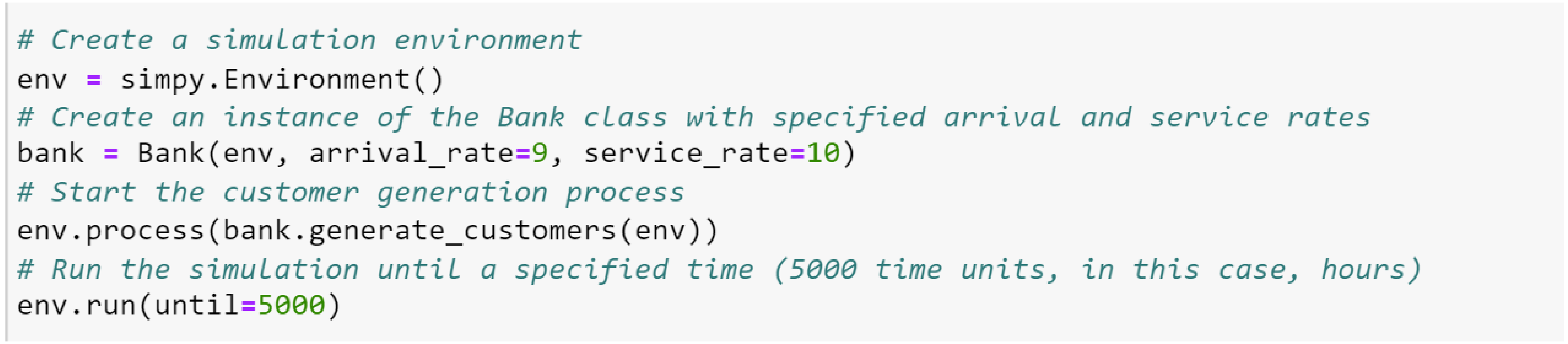
3) Generate customers arriving to the bank at random



```
# Define a method to generate customers arriving at the bank
    def generate_customers(self, env):
        while True:  # Run indefinitely
            interarrival_time = random.expovariate(self.arrival_rate)  # Generate interarrival time
            yield env.timeout(interarrival_time)  # Simulate time until the next customer arrives
            env.process(self.customer(env))  # Start a new customer process
```

M/M/1 Queue Simulation

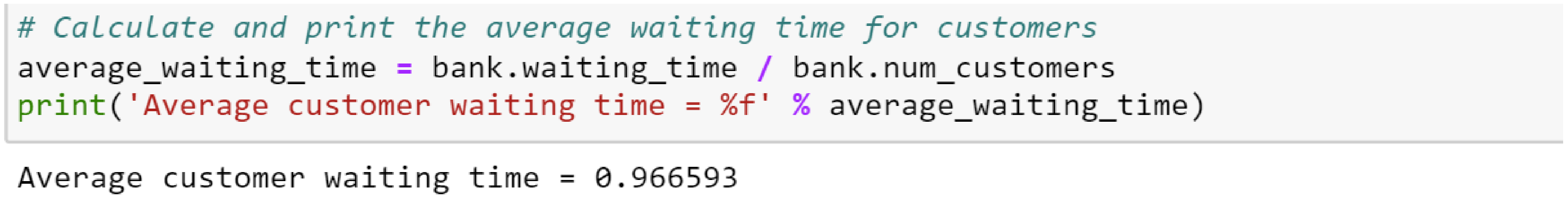
4) Run the Simulation



```
# Create a simulation environment
env = simpy.Environment()
# Create an instance of the Bank class with specified arrival and service rates
bank = Bank(env, arrival_rate=9, service_rate=10)
# Start the customer generation process
env.process(bank.generate_customers(env))
# Run the simulation until a specified time (5000 time units, in this case, hours)
env.run(until=5000)
```

M/M/1 Queue Simulation

5) Calculate the average waiting time and print it



```
# Calculate and print the average waiting time for customers
average_waiting_time = bank.waiting_time / bank.num_customers
print('Average customer waiting time = %f' % average_waiting_time)
```

```
Average customer waiting time = 0.966593
```