Chapter 8

Simulation

During OOP modeling, we make assumptions on the system regarding the relationship between objects and data and use algorithms to represent their behavior. These models are just an approximation of real systems. Real systems include complex interactions usually hardly represented by exact analytical models. In these cases, systems' behavior must be simulated.

Simulations are used to generate data to obtain statistics of real systems. These statistics are used to make decisions on variables configuration that are relevant for systems' performance. For example, if we need to decide how many points of sale we must have in a supermarket, we can simulate customers arrival, products availability, shopping time, and also measure check out times. We can estimate the optimal number of points of sale that result in a desirable customers' check out time.

The main advantages of simulations are fast experimentation, cost, and risk reduction, design feedback, and data generation.

Simulations mainly depend on time and runtime. The former corresponds to the virtual clock that approximates the real time elapsed in the simulation. The latter represents the required computational time to carry out the simulation. In general, we want to simulate large amounts of time using a minimal amount of runtime.

Events occurrences are modeled using probability distributions to have more realistic simulations. For example, the arrival time of customers, or the customers' service time in a given store, can be modeled using an *exponential* distribution. For this kind of distribution, it is necessary to define the average rate of event occurrences. For instance, when a person arrives at a queue each 20 minutes, then this event has a distribution with a rate of 1/20. The following code shows an example of the use of the `expovariate` function to generate exponentially distributed times:

```python
#00_expovariate.py
```

```python

from random import expovariate

'''We added a basis time of 0.5 to prevent
time 0 returned by the distribution.'''

client_arrival_time = round(expovariate(1/20) + 0.5)
server_time_1 = round(expovariate(1/50) + 0.5)
server_time_2 = round(expovariate(1/50) + 0.5)

print(client_arrival_time)
print(server_time_1)
print(server_time_2)
```

```
29
53
26
```

8.1 Synchronous Simulation

It corresponds to one of the simpler ways of implementing a simulation. In this case, we divide the total simulation time into small intervals. At each interval, the program verifies all activities involved in the system. The general algorithm of this type of simulation is as follows:

```
while time simulation does not end do
    Increase the time by one unit
    if events occur in this time interval then
        Simulate events
    end if
end while
```

For instance, let's consider the case of modeling a car inspection station. This system operates as a queue, where the vehicles arrive randomly with probability P_c, and are processed by a station during a random amount of time. This type of problems is known as $M/M/k$ according to Kendal's notation. This notation defines that customers come to the system in a *Markovian* way (M), the service time in the queue is also *Markovian* (M), and there are k servers to attend each car in the waiting queue.

```python
# 01_synchronous.py

from collections import deque
import random


class Vehicle:
    """
    This class represent vehicles which arrives to the mechanical
    workshop
    """

    def __init__(self, vehicles):
        # When a new vehicle is created is chosen randomly incoming
        # vehicle type and the average time of service'''

        self.vehicle_type = random.choice(list(vehicles))
        self._review_time = round(
            random.expovariate(vehicles[self.vehicle_type]))

    @property
    def review_time(self):
        return self._review_time

    @review_time.setter
    def review_time(self, value):
        self._review_time = value

    def show_type(self):
        print("Being treated: {0} with an average time of {1} minutes"
              .format(self.vehicle_type, self.review_time))


```

```python
class WorkShop:
    """
    This class represent the review line in the workshop.
    """

    def __init__(self):
        self.current_task = None
        self.review_time = 0

    def busy(self):
        return self.current_task is not None

    def next_vehicle(self, vehicle):
        self.current_task = vehicle
        self.review_time = vehicle.review_time
        vehicle.show_type()

    def tick(self):
        if self.current_task is not None:
            self.review_time -= 1
            if self.review_time <= 0:
                self.current_task = None


def new_vehicle_arrive():
    """
    This function returns if arrive a new vehicle to queue. It is
    sampled from a uniform probability distribution. The method
    returns True if the value delivered by the random function is
    greater than a given value.
    """
    return random.random() >= 0.8


def technical_workshop(max_time, vehicles):
    """
     This function handles the process or technical service.
    """

    # Fix the random seed
    random.seed(10)

    # A WorkShop is created
    workshop = WorkShop()

    # Empty review line
    review_line = deque()

    # Waiting time
    waiting_times = []

    # The simulation cycle is defined until the maximum time in
    # minutes, each time t is increased is evaluated if a new
    # vehicle arrives at the review queue

    for t in range(max_time):
        if new_vehicle_arrive():
            review_line.append(Vehicle(vehicles))

        if not workshop.busy() and len(review_line) > 0:
            # Next vehicle is taken out from review queue
            curr_vehicle = review_line.popleft()
            waiting_times.append(curr_vehicle.review_time)
            workshop.next_vehicle(curr_vehicle)

        # Decrease one tick of time to waiting vehicle
        workshop.tick()

    average_time = sum(waiting_times) / len(waiting_times)
    total_time = sum(waiting_times)
    print('Statistics:')
    print('Average waiting time {0:6.2f} min.'.format(average_time))
    print('Total workshop service time was', '{0:6.2f} min'.format(
        total_time))
    print('Total vehicles serviced: {0}'.format(len(waiting_times)))


if __name__ == '__main__':

    # The types of vehicles and the average service time are defined
    vehicles = {'motorcycle': 1.0 / 8, 'car': 1.0 / 15,
                'pickup_truck': 1.0 / 20}
    maximum_time = 200
    technical_workshop(maximum_time, vehicles)
```

Output:

```
Being treated: pickup_truck with an average time of 5 minutes
Being treated: car with an average time of 5 minutes
Being treated: motorcycle with an average time of 36 minutes
Being treated: motorcycle with an average time of 8 minutes
Being treated: motorcycle with an average time of 1 minutes
Being treated: car with an average time of 8 minutes
Being treated: pickup_truck with an average time of 6 minutes
Being treated: motorcycle with an average time of 9 minutes
Being treated: motorcycle with an average time of 14 minutes
Being treated: car with an average time of 2 minutes
Being treated: car with an average time of 43 minutes
Being treated: car with an average time of 9 minutes
Being treated: pickup_truck with an average time of 33 minutes
Being treated: car with an average time of 7 minutes
Being treated: pickup_truck with an average time of 20 minutes
Statistics:
Average waiting time  13.73 min.
Total workshop service time was 206.00 min
Total vehicles serviced: 15
```

Synchronous simulations require a lot of running time to produce results. Most of the time steps in the main simulation loop do not produce changes in the system. Verification of system's states and simulation constraints generates a waste of CPU time. Due to these downsides, in this chapter, we focus on *Discrete Event Simulation (DES)*.

8.2 Discrete Event Simulation (DES)

In DES paradigm exists a discrete sequence of events distributed in time, in which each event occurs at a determined instant t that generates a change in system's state. In contrast with the synchronous simulation, DES assumes that there are no variations in the system's states between consecutive events. This assumption allows us to jump directly to the next event, without wasting runtime. On each iteration, the simulation selects the next event by choosing the one that occurs first, according to its simulated time. The following pseudocode shows a general discrete-based event simulation algorithm:

```
while the events queue is not empty and the simulation time is not over do
    select the next event from the queue
    move the simulation time to the previously selected event's time
    simulate the event
end while
```

DES Model components

The following elements comprise a simulation model:

- A set of state variables that describe the system at any time. For example:

  A clock that stores the simulation time.

  A set of possible events, including the next instant they will take place.

- A set of simulation elements, for example:

  A method that controls the flow of different events.

  A set of performance variables, useful to keep simulation statistics.

Now we present an example of a technical car inspection station. Figure 8.1 shows the workflow of the system.

```
# 02_DES.py

```

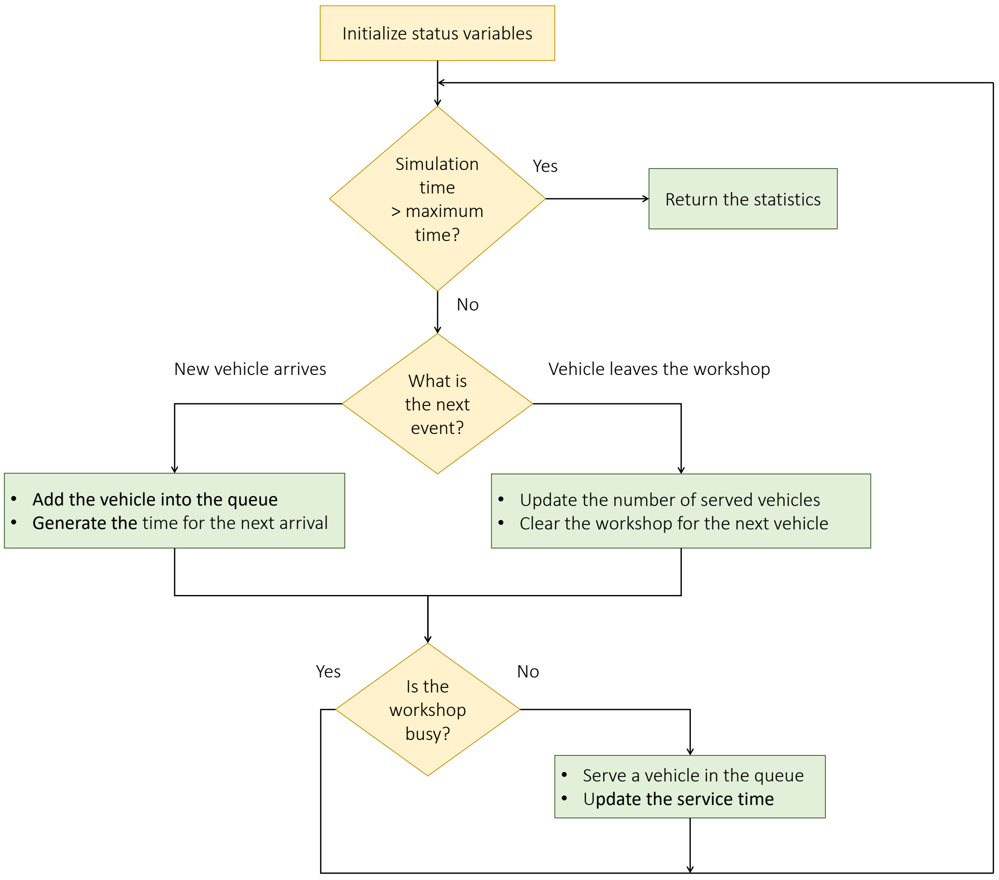
Initialize status variables
Simulation time > maximum time?
Yes
Return the statistics
No
New vehicle arrives
What is the next event?
Vehicle leaves the workshop
Add the vehicle into the queue
Generate the time for the next arrival
Update the number of served vehicles
Clear the workshop for the next vehicle
Yes
Is the workshop busy?
No
Serve a vehicle in the queue
Update the service time

Figure 8.1: The figure shows a flow chart of the technical car inspection example. The gray rectangles describe the statistics updated on each event. The green boxes represent the simulation events. Decisions are represented by yellow diamonds.

```python
from collections import deque
import random


class Vehicle:
    """
    This class represent vehicles which arrives to
    the mechanical workshop
    """

    def __init__(self, arrival_time=0):
        self.vehicle_type = random.choice(['motorcycle', 'pickup_truck', 'car'])
        self.arrival_time = arrival_time

    def __repr__(self):
        return 'Vehicle type: {0}'.format(self.vehicle_type)


class WorkShop:
    """
    This class represents the workshop and its behaviors.
    """

    def __init__(self, types):
        self.current_task = None
        self.review_time = 0
        self.types = types

    def pass_vehicle(self, vehicle):
        self.current_task = vehicle

        # Create a random review time
        current_type_rate = self.types[vehicle.vehicle_type]

        # We add 0.5 to avoid random times equals to zero
        self.review_time = round(random.expovariate(current_type_rate) + 0.5)

    @property
    def busy(self):
        return self.current_task is not None


class Simulation:
    """
    This class implements the simulation.
    Also you can use a function like in the previous case.
    All variables used in the simulation are initialized.
    """

    def __init__(self, maximum_time, arrival_rate, types):
        self.maximum_sim_time = maximum_time
        self.arrival_rate = arrival_rate
        self.simulation_time = 0
        self.next_vehicle_time = 0
        self.final_service_time = float('Inf')
        self.waiting_time = 0
        self.workshop = WorkShop(types)
        self.waiting_line = deque()
        self.served_vehicles = 0

    def next_vehicle(self, arrival_rate):
        # Update the arrival time of the next vehicle. We add 0.5 to avoid
        # arrivals time equals to zero.
        self.next_vehicle_time = self.simulation_time + \
                                 round(random.expovariate(arrival_rate) + 0.5)

    def run(self):
        """
        This method executes the simulation of the
        workshop and the waiting line
        """
        random.seed(10)

        self.next_vehicle(self.arrival_rate)

        # The cycle is executed verified the simulation time is less than
        # maximum simulation time
        while self.simulation_time < self.maximum_sim_time:

            # Update simulation time. Note that when the workshop is
            self.simulation_time = min(self.next_vehicle_time,
                                       self.final_service_time) if \
                    self.workshop.busy else self.next_vehicle_time

            print('[SIMULATION] time = {0} min'.format(self.simulation_time))

            # First, review the next event between arrival and the final of a
            # service
            if self.simulation_time == self.next_vehicle_time:

                # If a vehicle has arrived we have to add it to the queue,
                # and to generate the next arrival.
                self.waiting_line.append(Vehicle(self.simulation_time))
                self.next_vehicle(self.arrival_rate)

                print('[QUEUE] {0} arrives in: {1} min.'.format(
                    self.waiting_line[-1].vehicle_type,
                    self.simulation_time))


            elif self.simulation_time == self.final_service_time:

                print('[W_SHOP] Departure: {0} at {1} min.'.format(
                    self.workshop.current_task.vehicle_type,
                    self.simulation_time))

                self.workshop.current_task = None
                self.served_vehicles += 1

            # If the workshop is busy, the vehicle has to wait for its turn,
            # else can be served.

            if not self.workshop.busy and len(self.waiting_line) > 0:
                # Get the next vehicle in the waiting line
                next_vehicle = self.waiting_line.popleft()

                # The vehicle begin to be served
```

```
                    self.workshop.pass_vehicle(next_vehicle)

                    # Update the waiting time, added 0 actually
                    self.waiting_time += self.simulation_time \
                                         - self.workshop.current_task.arrival_time

                    # The next final time is generated
                    self.final_service_time = self.simulation_time \
                                              + self.workshop.review_time

                    print('[W_SHOP] {0} enters with a expected service time {'
                          '1} min.'.format(
                        self.workshop.current_task.vehicle_type,
                        self.workshop.review_time))

        print('Statistics:')
        print('Total service time {0} min.'.format(self.final_service_time))
        print('Total number of served vehicles: {0}'
              .format(self.served_vehicles))
        w_time = self.waiting_time / self.served_vehicles
        print('Average waiting time {0} min.'.format(round(w_time)))


if __name__ == '__main__':
    # Set the arrival rate in 5 minutes.
    arrival_rate_vehicles = 1 / 5

    # Here we define different types of vehicles and the their service time.
    vehicles = {'motorcycle': 1.0 / 8, 'car': 1.0 / 15,
                'pickup_truck': 1.0 / 20}

    # The simulation runs until 50 minutes.
    s = Simulation(70, arrival_rate_vehicles, vehicles)
    s.run()
```

```
[SIMULATION] time = 5 min
```

```
[QUEUE] pickup_truck arrives in: 5 min.
[W_SHOP] pickup_truck enters with a expected service time 1 min.
[SIMULATION] time = 6 min
[W_SHOP] Departure: pickup_truck at 6 min.
[SIMULATION] time = 9 min
[QUEUE] pickup_truck arrives in: 9 min.
[W_SHOP] pickup_truck enters with a expected service time 35 min.
[SIMULATION] time = 18 min
[QUEUE] car arrives in: 18 min.
[SIMULATION] time = 27 min
[QUEUE] motorcycle arrives in: 27 min.
[SIMULATION] time = 31 min
[QUEUE] pickup_truck arrives in: 31 min.
[SIMULATION] time = 32 min
[QUEUE] car arrives in: 32 min.
[SIMULATION] time = 44 min
[W_SHOP] Departure: pickup_truck at 44 min.
[W_SHOP] car enters with a expected service time 1 min.
[SIMULATION] time = 45 min
[W_SHOP] Departure: car at 45 min.
[W_SHOP] motorcycle enters with a expected service time 16 min.
[SIMULATION] time = 61 min
[QUEUE] car arrives in: 61 min.
[SIMULATION] time = 61 min
[W_SHOP] Departure: motorcycle at 61 min.
[W_SHOP] pickup_truck enters with a expected service time 11 min.
[SIMULATION] time = 64 min
[QUEUE] car arrives in: 64 min.
[SIMULATION] time = 66 min
[QUEUE] motorcycle arrives in: 66 min.
[SIMULATION] time = 72 min
[QUEUE] car arrives in: 72 min.
Statistics:
Total service time 72 min.
Total number of served vehicles: 4
```

```
Average waiting time 18 min.
```

As we can see, the variation of the simulation time depends exclusively on the events that occur during the simulation. On each iteration, a vehicle gets into the waiting line, and it randomly generates the arrival time for the next car (or event). Then, each time a vehicle enters the station, it makes a change in the state of the system. In case the workshop is not busy the next car is served during a random service time. In the opposite case, while the station is busy the incoming vehicles accumulate in the queue. When a car leaves the workshop, the simulation time is updated, generating a new change in the system's state.

8.3 Hands-On Activities

Activity 8.1

The first branch office of **Seguritas Bank** has two tellers to attend all clients. Each teller has its queue. Clients arriving are placed in the shortest line, if both lines are equal, they prefer the teller 1. When a customer finishes his visit and leaves the cashier, the last client of the other queue checks if he can improve his position by changing between lines, in that case, he moves to the other line instantly. Assume that all clients arrive in a random time between one and three minutes. Also, each teller takes a random time between one and ten minutes to serve one client. You must use DES to simulate this situation during eighty minutes. Do not forget to identify the states variables and the relevant events.

Activity 8.2

The zoo *GoodZoo* is thinking about creating a new exhibition where they will show the life cycle in a natural environment as a simulation. Six species are interacting in the environment: Tiger, Elephants, Jaguars, Penguins, Grass and Cephalopods. Their interactions follow the rules of the food chain. They ask you to create the simulation with the following events:

- **Feeding rules** Animals should eat according to their `diet`. To eat, they have to wait a random time within a range defined in the variable `time_for_food`. After an animal selects a prey from the ecosystem, it verifies if the victim belongs to its diet. After eating, animals get `food_energy`. In case the prey is not included in the animal's diet, it loses half of `food_energy` and tries to eat in the next simulation time.

- **Birth** New animals born at a rate of `new_animal`. Parents lose an amount of energy defined in the variable `giving_birth_energy`. After the birth, you should verify that parents have enough energy to stay alive.

- **Deaths** An animal can die for three reasons: it has reached its `life_expectancy`, another animal has eaten it, and its total energy has got to zero.

The simulation parameters are in the table below. `food_frequency` is uniformly distributed within a range specified in the table. The time between births (`new_animal`) follows an exponential distribution where λ is also specified in the table:

Parameters	**Tiger**	**Elephant**	**Jaguar**	**Penguin**
`food`	Elephant, Jaguar, Penguin	Grass	Elephant, Tiger, Penguin	Cephalopod
`food_frequency`	$(20, 30)$	$(8, 15)$	$(35, 55)$	$(4, 15)$
`food_energy`	30	4	20	5
λ	$\frac{1}{75}$	$\frac{1}{200}$	$\frac{1}{80}$	$\frac{1}{80}$
`new_animal_energy`	15	7	10	10
`life_expectancy`	300	500	350	90
`initial number of animals`	5	8	5	12

The simulation never runs out of Grass and Cephalopods, they do not perform any action, and only Elephants and penguins eat them. Their quantities will **always** be:

- Grass: initial number of elephants $\times 3$
- Cephalopods: initial number of penguins $\times 5$

The maximum simulation time is 1000 units of time, and you must run 1000 of these simulations to calculate and show the following statistics:

1. Average simulation time.
2. How many times each species becomes extinct.
3. When does a species becomes extinct.
4. How many animals of each species were born.
5. How many animals of each species were eaten.
6. How many animals of each species ran out of energy.
7. How many animals of each species died old.
8. Average lifetime per species.
9. How many individuals of each species were alive at the end of the simulations.
10. How many animals of each species had to wait one turn or more to find their food.
11. Average food-waiting-time per species.

Requirements

- You must use Discrete Events Simulations because an iterative simulation won't be fast enough to test all the cases that *GoodZoo* wants to tests.
- Animals can be part of many events at the same time. The verification order of events it is not important, but you have to make sure that animals that die at event n cannot exist at $n + 1$.
- Animals become part of the simulation the next instant of time after they born. That means that if they are born at t, they cannot be food for any animal that eats at t, and they cannot die or find food. These actions can be performed only from $t + 1$.

Notes

- Consider making a detailed model; it will get your job much easier

Here we summarize all what you have to do:

- Simulate feeding
  - Randomly find prey. If the chosen animal cannot be eaten (it is not part of the diet) then wait for one instant.
  - Check energy level and simulate death if that corresponds
  - Calculate statistics 1, 2, 3, 8, 9, 10, 11 and the deaths-counts per species.
  - Compute the next feeding time event
- Simulate births
  - Add a new animal to the simulation
  - Simulate the energy loss and deaths if corresponds
  - Calculate statistic 4
  - Compute next birth time event
- Simulate deaths
  - Delete an animal from the ecosystem when it has reached its maximum age.
  - Calculate statistics 5, 6 and 7