Лабораторна робота №3

Моделювання кінотеатру за допомогою SimPy

Уявіть, що вас найняли допомагати менеджеру невеликого місцевого кінотеатру. Кінотеатр отримує погані відгуки через довгий час очікування. Менеджер, який стурбований як витратами, так і задоволеністю клієнтів, може дозволити собі утримувати у штаті лише стільки працівників.

Менеджер особливо стурбований тим, який хаос може розгорнутися, коли почнуться покази блокбастерів: черги навколо кінотеатру, працівники, що працюють на межі своїх можливостей, розлючені глядачі, які пропускають початкові сцени... Такої ситуації, безумовно, не можна допустити!

Перевіривши відгуки, менеджер зміг визначити, що середньостатистичний кіноглядач у їхньому кінотеатрі готовий витратити максимум 10 хвилин з моменту приходу до моменту, коли він вмоститься в крісло. Іншими словами, середній час очікування в кінотеатрі повинен становити не більше 10 хвилин. Менеджер звернувся до вас за допомогою, щоб знайти рішення, як зменшити час очікування до 10 хвилин.

Мозковий штурм алгоритму моделювання

Перш ніж ви напишете хоча б один рядок коду, важливо спочатку зрозуміти, як ваш процес буде працювати в реальному житті. Це необхідно для того, щоб переконатися, що коли ви передасте його машині, процес буде точним відображенням того, що клієнти дійсно відчують. Ось як ви можете продумати кроки, які міг би зробити кіноглядач, щоб записати свій алгоритм:

1. **Прийти** до театру, стати в чергу і чекати, щоб придбати квиток.
2. **Купуєте** квиток у касі.
3. **Зачекайте** в черзі на перевірку квитка.
4. **Пройдіть** перевірку квитка у швейцара.
5. **Вибирають**, чи ставати в чергу до кіоску з їжею, чи ні:
    - **Якщо вони стають у чергу**, то купують їжу.
    - **Якщо не ставати в чергу,** то переходити до останнього кроку.
6. **Знайдіть** їхнє місце.

Це покрокова ітерація для кіноглядача, який купує квиток у касі кінотеатру. Ви вже бачите, якими частинами цього процесу можна керувати. Ви можете вплинути на тривалість очікування, збільшивши кількість касирів у касі.

Є також частини процесу, які не піддаються контролю, наприклад, найперший крок. Ви не можете контролювати, скільки клієнтів прийде або як швидко вони це зроблять. Ви можете зробити припущення, але не можете просто вибрати число, тому що це буде поганим відображенням реальності. Для цього параметра найкраще, що ви можете зробити, це використати наявні дані, щоб визначити відповідний час прибуття.

Setting Up the Environment

Before you start building your simulation, you need to make sure that your development environment is properly configured. The very first thing you'll want to do is import the necessary packages. You can do this by declaring import statements at the top of your file:

```
import simpy
import random
import statistics
```

These are the main libraries you'll use to build a script for the theater manager. Remember, the goal is to find the optimal number of employees that gives an average wait time of less than 10 minutes. To do this, you'll need to collect the length of time that it takes for each moviegoer to make it to their seats. The next step is to declare a list to hold these times:

```
wait_times = []
```

This list will contain the total amount of time each moviegoer spends moving through the theater, from arrival to sitting in their seat. You declare this list at the very top of the file so that you can use it inside any function you define later on.

Creating the Environment: Class Definition

The first part of the simulation you'll want to build is the blueprint for the system. This is going to be the overall environment inside which things happen, and people or objects move from one place to another. Remember, an **environment** can be one of many different systems, like a bank, a car wash, or a security checkpoint. In this case, the environment is a movie theater, so that will be the name of your class:

```
class Theater(object):
    def __init__(self):
        # More to come!
```

Now it's time to think through the parts of a **movie theater**. Of course, there's the theater itself, which is what you've called your environment. Later, you'll explicitly declare the theater as an actual environment using one of the simpy functions. For now, call it env for short and add it to the class definition:

```
class Theater(object):
    def __init__(self, env):
        self.env = env
```

Alright, what else might there be in a theater? You can figure this out by thinking through the simulation algorithm you planned out earlier. When a moviegoer arrives, they'll need to get in line at the box office, where a cashier will be waiting to help them out. Now you've discovered two things about the theater environment:

1. There are **cashiers**.
2. Moviegoers can **purchase tickets** from them.

Cashiers are a **resource** that the theater makes available to its customers, and they help moviegoers through the **process** of purchasing a ticket. Right now, you don't know how many cashiers are available in the simulated theater. In fact, that's the very problem you're trying to solve. How do wait times change, depending on the number of cashiers working on a given night?

You can go ahead and call this unknown variable num_cashiers. The exact value this variable will take can be sorted out later. For now, just know that it's an indispensable part of the theater environment. Add it to the class definition:

```
class Theater(object):
    def __init__(self, env, num_cashiers):
        self.env = env
        self.cashier = simpy.Resource(env, num_cashiers)
```

Here, you add the new parameter num_cashiers to your __init__() definition. Then, you create a resource self.cashier and use simpy.Resource() to declare how many can be in this environment at any given time.

Note: In simpy, resources are the parts of the environment (env) that are limited in number. Using one of them takes time, and only so many (num_cashiers) are available to be used at once.

There's one more step that you'll need to take. A cashier isn't going to purchase a ticket for *themselves*, right? They're going to help the moviegoer! Again, you know that this process of purchasing a ticket is going to take a certain amount of time. But just how much time?

Say you've asked the manager for historical data on the theater, like employee performance reviews or ticket purchase receipts. Based on this data, you've learned that it takes, on average, between 1 and 2 minutes to issue a ticket at the box office. How do you get simpy to mimic this behavior? It only takes one line of code:

```
yield self.env.timeout(random.randint(1, 3))
```

env.timeout() tells simpy to trigger an event after a certain amount of time has passed. In this case, the event is that a ticket was purchased.

The time this takes could be one minute, two minutes, or three minutes. You want each moviegoer to spend a different amount of time at the cashier. To do this, you use random.randint() to

choose a random number between the given low and high values. Then, for each moviegoer, the simulation will wait for the chosen amount of time.

Let's wrap this up in a tidy function and add it to the class definition:

```
class Theater(object):
    def __init__(self, env, num_cashiers):
        self.env = env
        self.cashier = simpy.Resource(env, num_cashiers)

    def purchase_ticket(self, moviegoer):
        yield self.env.timeout(random.randint(1, 3))
```

The one initiating the event in purchase_ticket() is the moviegoer, so they must be passed as a required argument.

You've selected a time-bound resource, defined its related process, and codified this in your class definition. For this tutorial, there are two more resources you'll need to declare:

1. **Ushers** to check tickets
2. **Servers** to sell food

After checking the data the manager sent over, you determine that servers take anywhere between 1 and 5 minutes to complete an order. In addition, ushers are remarkably fast at checking tickets, with an average speed of 3 seconds!

You'll need to add these resources to your class and define the corresponding functions check_ticket() and sell_food(). Can you figure out what the code should look like? When you've got an idea, you can expand the code block below to check your understanding:

simulate.pyShow/Hide

Take a close look at the new resources and functions. Notice how they follow the same format as described above. sell_food() uses random.randint() to generate a random number between 1 and 5 minutes, representing the time it would take a moviegoer to place an order and receive their food.

The time delay for check_ticket() is a bit different because the ushers only take 3 seconds. Since simpy works in minutes, this value needs to be passed as a fraction of a minute, or 3 / 60.

Moving Through the Environment: Function Definition

Alright, you've set up the environment by defining a class. You have resources and processes. Now you need a **moviegoer** to use them. When a moviegoer arrives at the theater, they'll request a resource, wait for its process to complete, and then leave. You'll create a function, called go_to_movies(), to keep track of this:

```
def go_to_movies(env, moviegoer, theater):
    # Moviegoer arrives at the theater
    arrival_time = env.now
```

There are three arguments passed to this function:

1. **env:** The moviegoer will be controlled by the environment, so you'll pass this as the first argument.
2. **moviegoer:** This variable tracks each person as they move through the system.
3. **theater:** This parameter gives you access to the processes you defined in the overall class definition.

You also declare a variable arrival_time to hold the time at which each moviegoer arrives at the theater. You can get this time using the simpy call to env.now.

You'll want each of the processes from your theater to have corresponding **requests** in go_to_movies(). For example, the first process in the class is purchase_ticket(), which uses a cashier resource. The moviegoer will need to make a request to the cashier resource to help them through the purchase_ticket() process. Here's a table to summarize this:

Process in theater	Request in go_to_movies()
purchase_ticket()	Request a cashier
check_ticket()	Request an usher

Process in theater	Request in go_to_movies()
sell_food()	Request a server

The cashier is a shared resource, which means that many moviegoers will use the same cashier. However, a cashier can only help one moviegoer at a time, so you'll need to include some waiting behavior in your code. Here's how that works:

```
def go_to_movies(env, moviegoer, theater):
    # Moviegoer arrives at the theater
    arrival_time = env.now

    with theater.cashier.request() as request:
        yield request
        yield env.process(theater.purchase_ticket(moviegoer))
```

Here's how this code works:

1. **theater.cashier.request():** moviegoer generates a request to use a cashier.
2. **yield request:** moviegoer waits for a cashier to become available if all are currently in use. To learn more about the yield keyword, check out How to Use Generators and yield in Python.
3. **yield env.process():** moviegoer uses an available cashier to complete the given process. In this case, that's to purchase a ticket with a call to theater.purchase_ticket().

After a resource is used, it must be freed up for the next agent to use. You could do this explicitly with release(), but in the code above, you use a with statement instead. This shortcut tells the simulation to automatically release the resource once the process is complete. In other words, once the ticket is bought, the moviegoer will leave, and the cashier will automatically be ready to take the next customer.

When a cashier is freed up, the moviegoer will spend some time buying their ticket. env.process() tells the simulation to go to the Theater instance and run the purchase_ticket() process on this moviegoer. The moviegoer will repeat this **request, use, release** cycle to have their ticket checked:

```
def go_to_movies(env, moviegoer, theater):
    # Moviegoer arrives at the theater
    arrival_time = env.now

    with theater.cashier.request() as request:
        yield request
        yield env.process(theater.purchase_ticket(moviegoer))

    with theater.usher.request() as request:
        yield request
        yield env.process(theater.check_ticket(moviegoer))
```

Here, the structure for the code is the same.

Then, there's the optional step of buying food from the concession stand. You can't know whether a moviegoer will want to purchase snacks and drinks. One way to deal with this uncertainty is to introduce a bit of **randomness** to the function.

Each moviegoer either will or will not want to buy food, which you can store as the Boolean values True or False. Then, use the random module to have the simulation randomly decide whether or not *this* particular moviegoer is going to proceed to the concession stand:

```
def go_to_movies(env, moviegoer, theater):
    # Moviegoer arrives at the theater
    arrival_time = env.now

    with theater.cashier.request() as request:
        yield request
        yield env.process(theater.purchase_ticket(moviegoer))

    with theater.usher.request() as request:
```

```
            yield request
            yield env.process(theater.check_ticket(moviegoer))

        if random.choice([True, False]):
            with theater.server.request() as request:
                yield request
                yield env.process(theater.sell_food(moviegoer))
```

This conditional statement will return one of two outcomes:

1. **True:** The moviegoer will request a server and order food.
2. **False:** The moviegoer will instead go to find their seats without purchasing any snacks.

Now, remember the goal of this simulation is to determine the number of cashiers, ushers, and servers that should be on staff to keep wait times under 10 minutes. To do this, you'll need to know how long it took any given moviegoer to make it to their seats. You use env.now at the beginning of the function to track the arrival_time, and again at the end when each moviegoer is finished with all processes and heading into the theater:

```
def go_to_movies(env, moviegoer, theater):
    # Moviegoer arrives at the theater
    arrival_time = env.now

    with theater.cashier.request() as request:
        yield request
        yield env.process(theater.purchase_ticket(moviegoer))

    with theater.usher.request() as request:
        yield request
        yield env.process(theater.check_ticket(moviegoer))

    if random.choice([True, False]):
        with theater.server.request() as request:
            yield request
            yield env.process(theater.sell_food(moviegoer))

    # Moviegoer heads into the theater
    wait_times.append(env.now - arrival_time)
```

You use env.now to get the time at which the moviegoer has finished all processes and made it to their seats. You subtract the moviegoer's arrival_time from this departure time and append the resulting time difference to your waiting list, wait_times.

Note: You could store the departure time in a separate variable like departure_time, but this would make your code very repetitive, which violates the D.R.Y. principle.

This moviegoer is ready to watch some previews!

Making Things Happen: Function Definition

Now you'll need to define a function to run the simulation. run_theater() will be responsible for creating an instance of a theater and generating moviegoers until the simulation stops. The first thing this function should do is create an instance of a theater:

```
def run_theater(env, num_cashiers, num_servers, num_ushers):
    theater = Theater(env, num_cashiers, num_servers, num_ushers)
```

Since this is the main process, you'll need to pass all of the unknowns you've declared so far:

- num_cashiers
- num_servers
- num_ushers

These are all variables that the simulation needs to create and control the environment, so it's absolutely vital to pass them all. Then, you define a variable theater and tell the simulation to set up the theater with a certain number of cashiers, servers, and ushers.

You also might want to start your simulation with a few moviegoers waiting at the theater. There will probably be a few people ready to go as soon as the doors open! The manager says to expect around 3 moviegoers in line ready to buy tickets as soon as the box office opens. You can tell the simulation to go ahead and move through this initial group like so:

```
def run_theater(env, num_cashiers, num_servers, num_ushers):
    theater = Theater(env, num_cashiers, num_servers, num_ushers)

    for moviegoer in range(3):
        env.process(go_to_movies(env, moviegoer, theater))
```

You use range() to populate the theater with 3 moviegoers. Then, you use env.process() to tell the simulation to prepare to move them through the theater. The rest of the moviegoers will make it to the theater in their own time. So, the function should keep sending new customers into the theater as long as the simulation is running.

You don't know how long it will take new moviegoers to make it to the theater, so you decide to look at past data. Using timestamped receipts from the box office, you learn that moviegoers tend to arrive at the theater, on average, every 12 seconds. Now all you have to do is tell the function to wait this long before generating a new person:

```
def run_theater(env, num_cashiers, num_servers, num_ushers):
    theater = Theater(env, num_cashiers, num_servers, num_ushers)

    for moviegoer in range(3):
        env.process(go_to_movies(env, moviegoer, theater))

    while True:
        yield env.timeout(0.20)  # Wait a bit before generating a new person

        # Almost done!...
```

Note that you use the decimal number 0.20 to represent 12 seconds. To get this number, you simply divide 12 seconds by 60 seconds, which is the number of seconds in a minute.

After waiting, the function should increment moviegoer by 1 and generate the next person. The generator function is the same one you used to initialize the first 3 moviegoers:

```
def run_theater(env, num_cashiers, num_servers, num_ushers):
    theater = Theater(env, num_cashiers, num_servers, num_ushers)

    for moviegoer in range(3):
        env.process(go_to_movies(env, moviegoer, theater))

    while True:
        yield env.timeout(0.20)  # Wait a bit before generating a new person

        moviegoer += 1
        env.process(go_to_movies(env, moviegoer, theater))
```

That's it! When you call this function, the simulation will generate 3 moviegoers to start and begin moving them through the theater with go_to_movies(). After that, new moviegoers will arrive at the theater with an interval of 12 seconds and move through the theater in their own time.

Calculating the Wait Time: Function Definition

At this point, you should have a list wait_times that contains the total amount of time it took each moviegoer to make it to their seat. Now you'll want to define a function to help calculate the

average time a moviegoer spends from the time they arrive to the time they finish checking their ticket. get_average_wait_time() does just this:

```
def get_average_wait_time(wait_times):
    average_wait = statistics.mean(wait_times)
```

This function takes your wait_times list as an argument and uses statistics.mean() to calculate the average wait time.

Since you're creating a script that will be used by the movie theater manager, you'll want to make sure that the output can be read easily by the user. You can add a function called calculate_wait_time() to do this:

```
def calculate_wait_time(arrival_times, departure_times):
    average_wait = statistics.mean(wait_times)
    # Pretty print the results
    minutes, frac_minutes = divmod(average_wait, 1)
    seconds = frac_minutes * 60
    return round(minutes), round(seconds)
```

The last part of the function uses divmod() to return the results in minutes and seconds, so the manager can easily understand the program's output.

Choosing Parameters: User Input Function Definition

As you've built these functions, you've run into a few variables that have not been clearly defined:

- num_cashiers
- num_servers
- num_ushers

These variables are the parameters that you can **change** to see how the simulation changes. If a blockbuster movie has customers lining up around the block, how many cashiers should be working? What if people are flying through the box office but getting stuck at concessions? What value of num_servers will help ease the flow?

Note: That's the beauty of simulation. It allows you to try these things out so that you can determine the best possible decision in real life.

Whoever is using your simulation needs to be able to change the values of these parameters to try out different scenarios. To this end, you'll create a helper function to get these values from the user:

```
def get_user_input():
    num_cashiers = input("Input # of cashiers working: ")
    num_servers = input("Input # of servers working: ")
    num_ushers = input("Input # of ushers working: ")
    params = [num_cashiers, num_servers, num_ushers]
    if all(str(i).isdigit() for i in params):  # Check input is valid
        params = [int(x) for x in params]
    else:
        print(
            "Could not parse input. The simulation will use default values:",
            "\n1 cashier, 1 server, 1 usher.",
        )
        params = [1, 1, 1]
    return params
```

This function simply calls Python's input() function to retrieve data from the user. Because user input runs the risk of being messy, you can include an if/else clause to catch anything invalid. If the user inputs bad data, then the simulation will run with default values.

Finalizing the Setup: Main Function Definition

The last function you'll want to create is main(). This will ensure your script runs in the proper order when you execute it on the command line.

```
def main():
  # Setup
  random.seed(42)
  num_cashiers, num_servers, num_ushers = get_user_input()

  # Run the simulation
  env = simpy.Environment()
  env.process(run_theater(env, num_cashiers, num_servers, num_ushers))
  env.run(until=90)

  # View the results
  mins, secs = get_average_wait_time(wait_times)
  print(
     "Running simulation...",
     f"\nThe average wait time is {mins} minutes and {secs} seconds.",
  )
```

Here's how main() works:

1. **Set up** your environment by declaring a random seed. This ensures your output will look like what you see in this Lecture .
2. **Query** the user of your program for some input.
3. **Create** the environment and save it as the variable env, which will move the simulation through each time step.
4. **Tell** simpy to run the process run_theater(), which creates the theater environment and generates moviegoers to move through it.
5. **Determine** how long you want the simulation to run. As a default value, the simulation is set to run for 90 minutes.
6. **Store** the output of get_average_wait_time() in two variables, mins and secs.
7. **Use** print() to show the results to the user.

How to Run the Simulation

With just a few more lines of code, you'll be able to watch your simulation come to life. But first, here's an overview of the functions and classes you've defined so far:

- **Theater:** This class definition serves as a blueprint for the environment you want to simulate. It determines some information about that environment, like what kinds of resources are available, and what processes are associated with them.
- **go_to_movies():** This function makes explicit requests to use a resource, goes through the associated process, and then releases it to the next moviegoer.
- **run_theater():** This function controls the simulation. It uses the Theater class blueprint to create an instance of a theater, and then calls on go_to_movies() to generate and move people through the theater.
- **get_average_wait_time():** This function finds the average time it takes a moviegoer to make it through the theater.
- **calculate_wait_time():** This function ensures the final output is easy for the user to read.
- **get_user_input():** This function allows the user to define some parameters, like how many cashiers are available.
- **main():** This function ensures that your script runs properly in the command line.

Now, you only need two more lines of code to invoke your main function:

```
if __name__ == '__main__':
   main()
```

With that, your script is ready to run! Open up your terminal, navigate to where you've stored simulate.py, and run the following command:

```
$ python simulate.py
Input # of cashiers working:
```

You'll be prompted to select the parameters you want for your simulation. Here's what the output looks like with default parameters:

```
$ python simulate.py
Input # of cashiers working: 1
Input # of servers working: 1
Input # of ushers working: 1
Running simulation...
The average wait time is 42 minutes and 53 seconds.
```

Whoa! That's a long time to be waiting around!

When to Change Things Up

Remember, your goal is to approach the manager with a solution for how many employees he'll need on staff to keep wait times under 10 minutes. To this end, you'll want to play around with your parameters to see which numbers offer an optimal solution.

First, try something completely insane and max out the resources! Say there were 100 cashiers, 100 servers, and 100 ushers working in this theater. This is impossible, of course, but using insanely high numbers will quickly tell you what the system's limit is. Try it now:

```
$ python simulate.py
Input # of cashiers working: 100
Input # of servers working: 100
Input # of ushers working: 100
Running simulation...
The average wait time is 3 minutes and 29 seconds.
```

Even if you maxed out the resources, you would only get wait times down to 3 and a half minutes. Now try and change the numbers to see if you can get wait times down to 10 minutes, like the manager requested. What solution did you come up with? You can expand the code block below to see one possible solution:

simulate.pyShow/Hide

At this point, you would present your results to the manager and make a suggestion to help improve the theater. For instance, to cut down on costs, he might want to install 10 ticket kiosks at the front of the theater instead of keeping 10 cashiers on hand each night.

Завдання одне для всіх

1. Визначте, скільки касирів, офіціантів, швейцарів потрібно, щоб отримати середній час очікування менше 10 хв.
2. Припустимо, що кількість касирів дорівнює 10. Скільки потрібно офіціантів, швейцарів, обслуговуючого персоналу, щоб середній час очікування не перевищував 9 хв.
3. Визначте, скільки потрібно касирів, офіціантів, серверів, щоб середній час очікування був менше 5 хв.
4. Якщо у вас є 7 касирів, 10 швейцарів, 8 офіціантів. Який середній час очікування?

Індивідуальні завдання

1. Придумати приклад багатоканальної СМО з відмовами та дослідити її параметри (***для різних студентів приклади не мають повторюватись***)

   Вимоги до прикладів:

   1)Реалістичність прикладу і його параметрів (Наприклад, пропускна здатність заправки не може бути більше 100 000 за день чи кількість відвідувачів кінотеатру з одним залом не може перевищувати 5000 і т.д.)

   2) При виборі параметрів розподілів вхідного потоку і інтенсивності обслуговування теж користуватись реалістичністю і наближеність до природніх умов.

   3) Перед моделюванням узгодити приклад з викладачем, котрий веде лабораторні роботи

4) Під час дослідження параметрів СМО змінювати не лише параметри розподілів, алей розглядати зворотну задачу зміни параметрів СМО.