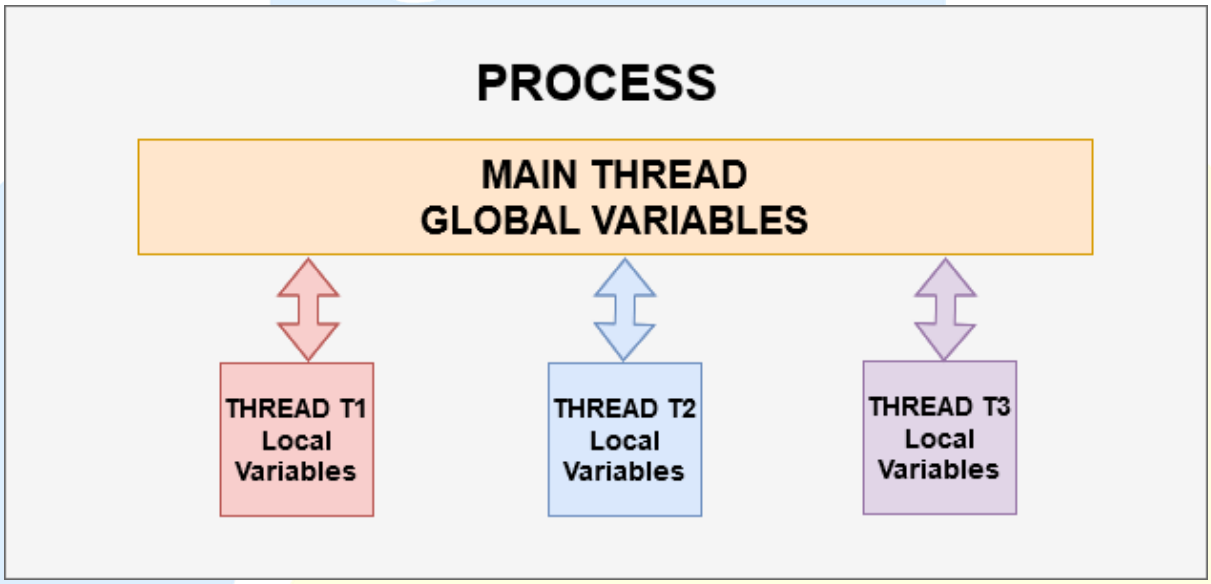
Threading in Python

Threading allows you to have different parts of your process run concurrently. These different parts are usually individual and have a separate unit of execution belonging to the same process. The process is nothing but a running program that has individual units that can be run concurrently. For example, A web-browser could be a process, an application running multiple cameras simultaneously could be a process; a video game is another example of a process.



Inside a process comes the concept of multiple threading or commonly known as multi-threading, where multiple threads work together to achieve a common goal. The most crucial benefit of using threads is that it allows you to run the program in parallel.

Let us understand the concept of threading with the help of an example. Imagine you have an application which counts the number of cars entering and exiting the mall's parking. Your apparatus has various cameras that monitor the entry and exit connecting to a central device. Each camera will have an algorithm to monitor the flow of cars, which will belong to the same process or program. However, each camera, along with the algorithm it is being run on, could be part of a separate thread. Not only that, but even the frames being read from the camera and the algorithm predicting on the frames could also be two separate threads.

Another example could be a video game in which the process has to run the tasks in parallel like the graphics, user interaction, and networking (while

playing multiplayer games) because it must be always responsive. And to accomplish this, it must make use of the concept of multi-threading, where each thread would be responsible for running each independent and individual task.

A thread has its flow of execution, which means that the process will have multiple things happening at one time.

Some advantages of having threading in your program:

- Multi-threading allows the program to speed up the execution if it has multiple CPUs.
- It also lets you perform other tasks while the I/O operations are being performed with the help of multiple threads or even main thread along with a single thread. For example, the speed at which the frames from the camera are read and inferred by the algorithm will be handled by different threads. Hence, the algorithm will not have to wait for the frame to be inputted, and the frame reading part will not have to wait for the algorithm execution to complete to be able to read the next frame.
- Threads within the same process can share the memory and resources of the main thread.

Threading in Python

- In Python, the threading module is a built-in module which is known as **threading** and can be directly imported.
- Since almost everything in Python is represented as an object, threading also is an object in Python. A thread is capable of:
    - Holding data,
    - Stored in data structures like dictionaries, lists, sets, etc.
    - Can be passed as a parameter to a function.
- A thread can also be executed as a process.
- A thread in Python can have various states like:
    - Wait
    - Locked

Implementing threading using the Threading module

Let us use the same example as you used above, but this time you will use the ***threading*** module instead of the ***_thread*** module.

```
import threading
import time


def thread_delay(thread_name, delay):
    count = 0
    while count < 3:
        time.sleep(delay)
        count += 1
        print(thread_name, '-------->', time.time())
```

In the Thread class constructor, you will pass in the target function thread_delay and the arguments of that function.

```
t1 = threading.Thread(target=thread_delay, args=('t1', 1))
t2 = threading.Thread(target=thread_delay, args=('t2', 3))
```

In the threading module, in order to run or execute the thread, you make use of the start() method, which simply responsible for running the thread.

```
t1.start()
t2.start()
```

Complete code

```
import threading
import time


def thread_delay(thread_name, delay):
    count = 0
    while count < 3:
        time.sleep(delay)
        count += 1
        print(thread_name, '-------->', time.time())


t1 = threading.Thread(target=thread_delay, args=('t1', 1))
t2 = threading.Thread(target=thread_delay, args=('t2', 3))

t1.start()
t2.start()
```

Output

```
t1 --------> 1667372277.9348533
t1 --------> 1667372278.9374816
t2 --------> 1667372279.9294157
t1 --------> 1667372279.9446812
t2 --------> 1667372282.9426105
t2 --------> 1667372285.9480076
```

You will also use the **join** method, which means that wait until all the thread execution is complete. So whatever code you have written after the **join** method will only be executed once these threads have terminated.

```
import threading
import time


def thread_delay(thread_name, delay):
    count = 0
    while count < 3:
        time.sleep(delay)
        count += 1
        print(thread_name, '-------->', time.time())


t1 = threading.Thread(target=thread_delay, args=('t1', 1))
t2 = threading.Thread(target=thread_delay, args=('t2', 3))

t1.start()
t2.start()

t1.join()
t2.join()

print("Thread execution is complete!")
```

Output

```
t1 --------> 1667372365.4909344
t1 --------> 1667372366.4995704
t2 --------> 1667372367.482059
t1 --------> 1667372367.5135112
t2 --------> 1667372370.4965212
t2 --------> 1667372373.5096905
Thread execution is complete!
```