

What is Parallelism?

Earlier machines used to have only one core within the CPU where all the processing used to take place.

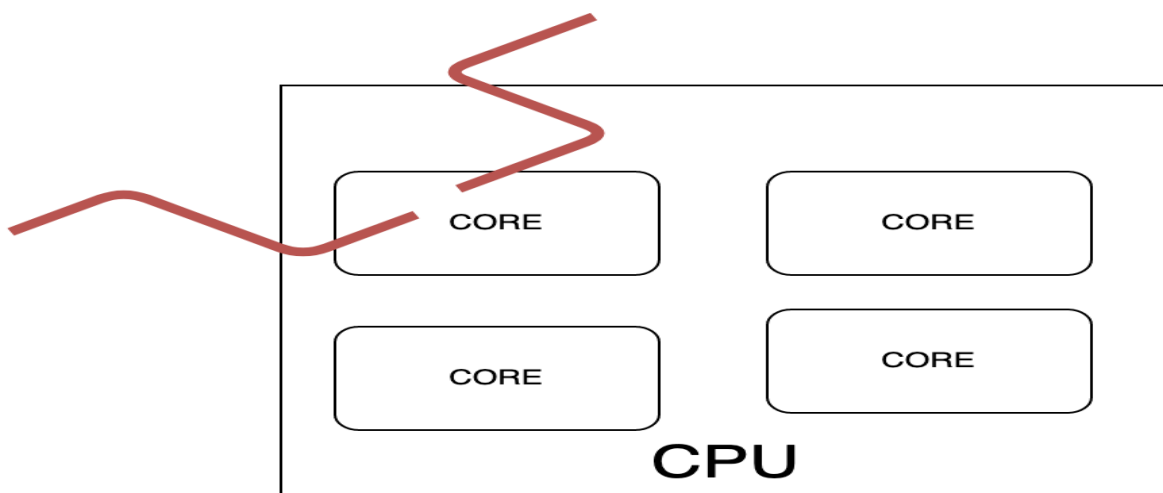
Why is the number of cores important- It is because it tells about the capacity of the machine to handle multiple things. If you have 16 cores, then you can do 16 different operations at the exact same time.

Let us say you want to perform 16 different addition operations and assume each operation takes 1 second. In a single-core machine, you have to perform these operations one by one, which means the 16 addition operations get done in 16 seconds. Now in a 16 core machine, you can deploy the 16 addition operations to each core at the same time and get the job done in 1 second. This is called **Parallelism**.

Threading

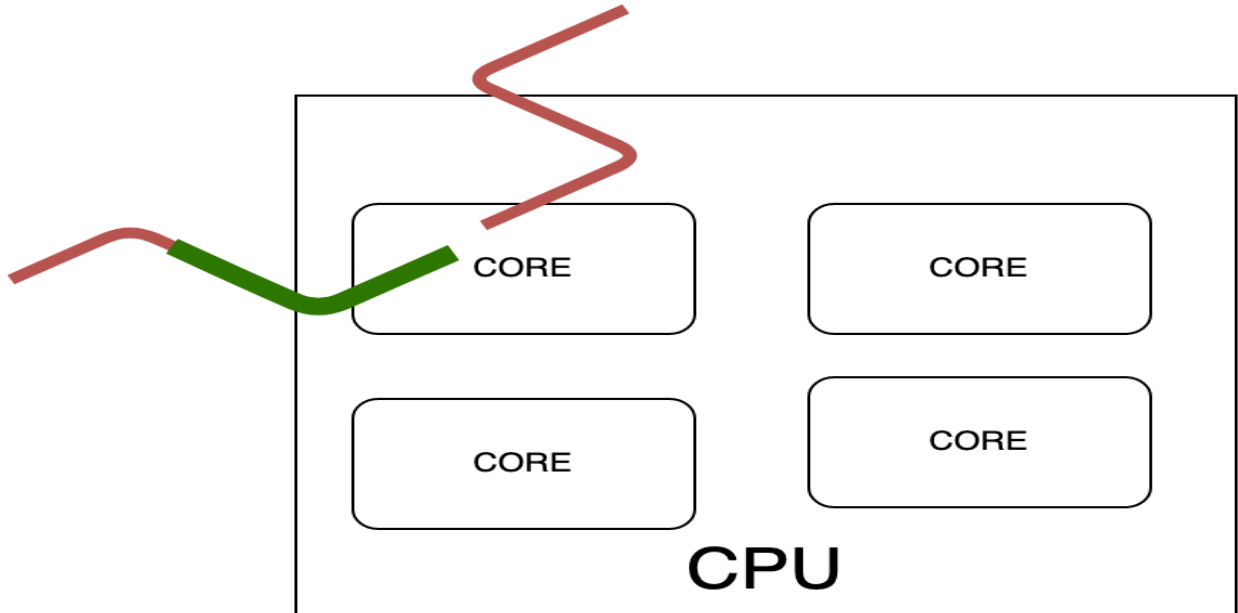
Thread is a set of operations that needs to execute. The thread will be deployed in one of the cores in the CPU. Note- 1 thread can be deployed only in 1 core, it cannot be transferred/switched to

Let us have deployed two threads to a core. **Note- A core can do only one thing at a time.**

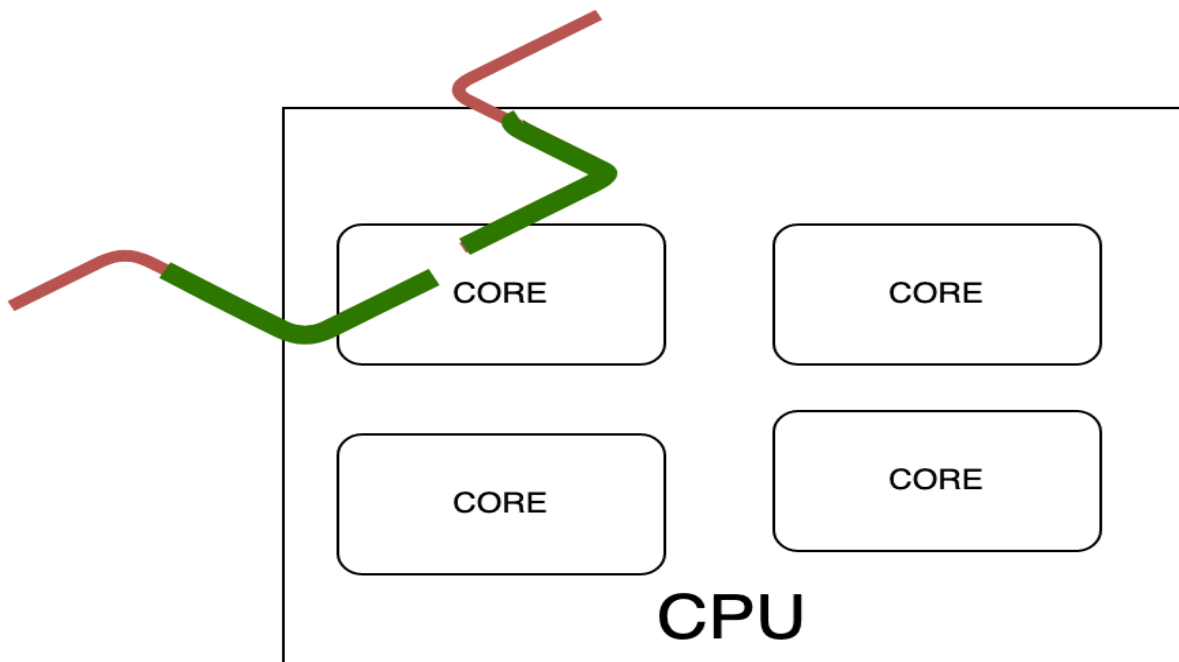


Now we can process the two threads in the way we want.

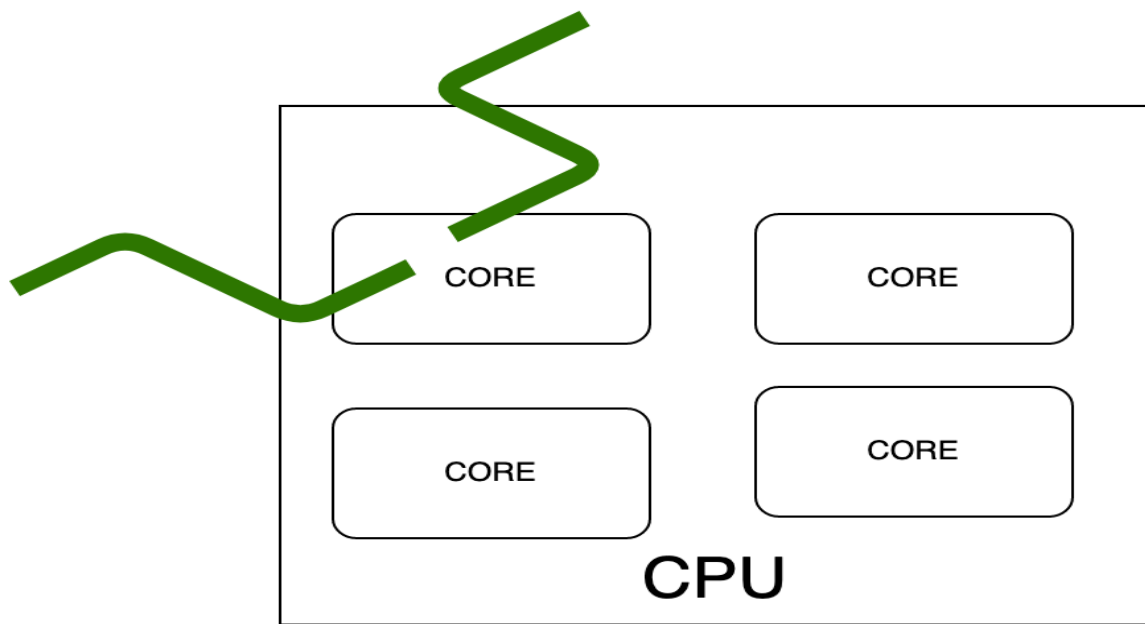
First, we can process half of the first thread.



Half of the next thread can be processed now.



The remaining half of the threads can be processed in a similar fashion.



This what threading is- It is how do we run different things on the same CPU core. **TLDR- Threading is about how do we handle the threads in a core.**

Note- Threading does not involve running on multiple cores. It is about how to sequence the set of programs(threads) in the same core. In the above example, we are gonna tell the CPU how to sequence and execute the two threads on the given core.

You can actually see the number of threads that are currently running on your machine. In Windows- Go to Task Manager → Performance → CPU.

Why do we need threading? Why can't we actually process one thread at a time and move on to the next?

Sometimes, a thread can go into a hanging, which means it is supposed to be idle at that point in time. The best example is `time.sleep()` function, which does nothing but waits for a given time. While one thread is idle/hanging, we can move on and process the other thread until the previous thread becomes active. **TLDR- When one thread is waiting, you can process the other thread meanwhile.**

A small example

Let us explain the threading with a small example. Look at the code snippet below.

#Part One of the code

```
import time
print(1)
time.sleep(10)
print('Done sleeping')
print(2)
```

#Part Two of the code

```
print(3)
```

When you execute the whole code as a single thread, the code is executed step by step. First, the library is imported. Then '1' is printed. The threads sleep for 10 seconds. Next '2' is printed followed by 'Done sleeping' and finally '3' printed.

Output-

```
1
Done sleeping
2
3
```

Now let us say you are executing the code as two threads. Part one as a thread and part two as a thread. (Note- By default, the Python code is not provisioned with threading- we need to import the **threading** library to do so.)

First, the library is imported, and then '1' is printed. Now the thread goes to sleep. This is where threading comes into action.

Thread 1 (Part 1)



Thread 2 (Part 2)

The core now switches to the other thread.

Thread 1 (Part 1)



Thread 2 (Part 2)

Now '3' is printed. Since all the process is done in Thread 2, the core now switches back to Thread 1 (which is still in sleep).

Thread 1 (Part 1)



Thread 2 (Part 2)

Now after the sleep duration, '2' is printed.

So the output will be

Output -

1

3

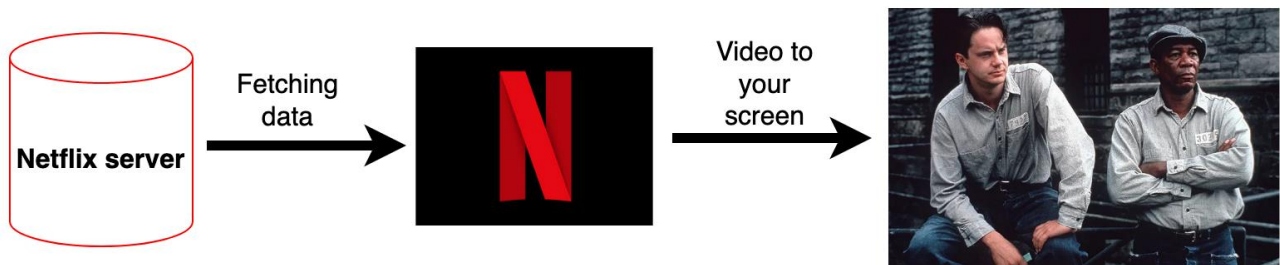
Done sleeping

2

Practical example

As always, a concept is only clear when we explain it with a real-world example. I/O processes are the ones that benefit from threading.

Let us say you are watching Shawshank Redemption on Netflix. Now two things happen while you are watching Andy Dufresne suffering in jail- One- the application fetches data from the server, Two- The fetched data is shown to you like a movie on your screen.



Imagine what would be the situation without threading. You would have to wait for the video to get downloaded once in a while, watch the segment that was fetched, wait for the next segment to get downloaded, and so on.

Thanks to threading, we can divide the two processes into different threads. While one thread fetches data (that is, it is in hang/sleep mode), the other thread can show you the amazing performance of Morgan Freeman.

It is also much useful for you as a Data Scientist. For example, when you scrape the data from multiple web pages, you can simply deploy them in multiple

threads and make it faster. Even when you push the data to a server, you can do so in multiple threads, so that when one thread is idle others can be triggered.

A detailed example

As said before, by default, the Python code is not provisioned with threading- we need to import the threading library to do so.

Take a look at the code.

```
import threading
import time

def sleepy_man(secs):
    print('Starting to sleep inside')
    time.sleep(secs)
    print('Woke up inside')

x = threading.Thread(target = sleepy_man, args = (1,))
x.start()
print(threading.activeCount())
time.sleep(1.2)
print('Done')
```

If you execute the above code, you will get the following output.

Output-

```
Starting to sleep inside
2
Woke up inside
Done
```

First, let me explain the code step by step. Then we will analyze the output.

- You import the library's **threading** and **time**. **threading** is the library that will allow us to create threads and **time** is the library that contains the function `sleep`.
- The function ***sleepy_man*** takes in the one argument- *secs*. It first prints 'Starting to sleep inside'. Then it sleeps for the *secs* seconds and then it prints 'Woke up inside'.
- This is the part where we start creating threads. We need to define by calling the class ***threading.Thread***. We need to pass two arguments- **target** which is the function block that needs to be threaded, **args** which are the arguments that need to be passed to the function. A thread object is returned which is now stored in *x*.

```
x = threading.Thread(target = sleepy_man, args = (10,))
```

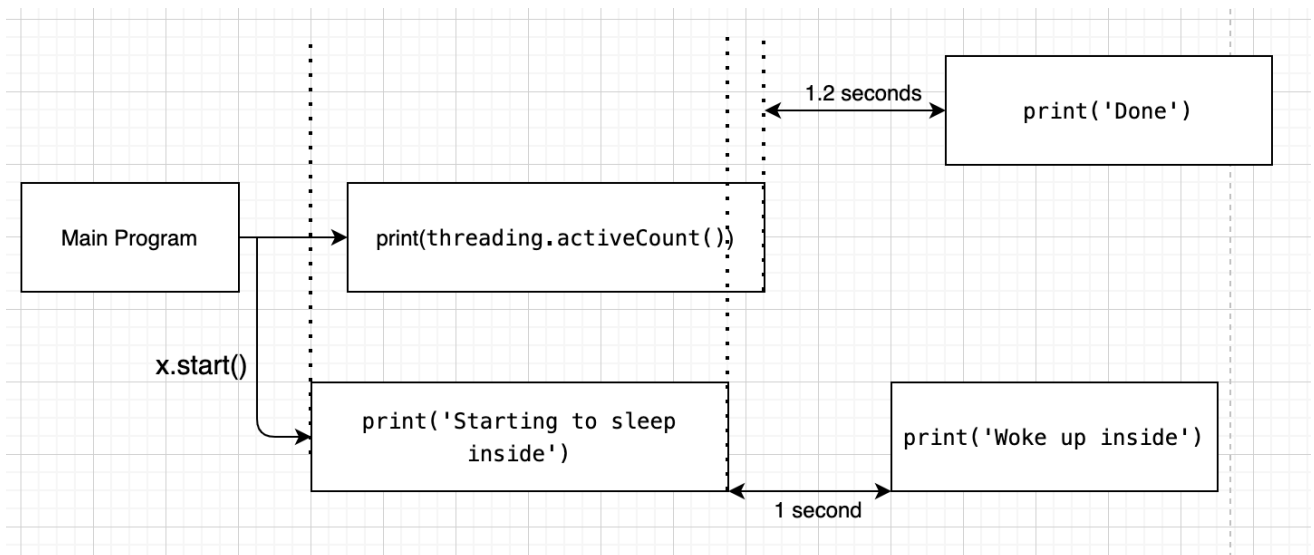
- Now after defining the thread class, we need to call the function ***start()*** so as to initiate the threading

```
x.start()
```

- Note- Now we have two threads. One default thread for the program and a new thread which we defined. Thus the active thread count is two.
- Thus the statement should print '2'.

```
print(threading.activeCount())
```

Now let us look at the flow of control. Once you call the `start()` method, it triggers `sleepy_man()` and it runs in a separate thread. The main program will also run in parallel as another thread. The flow is shown in the image below.



Now let us increase the time in which the program sleeps inside the function.

```
import threading
import time

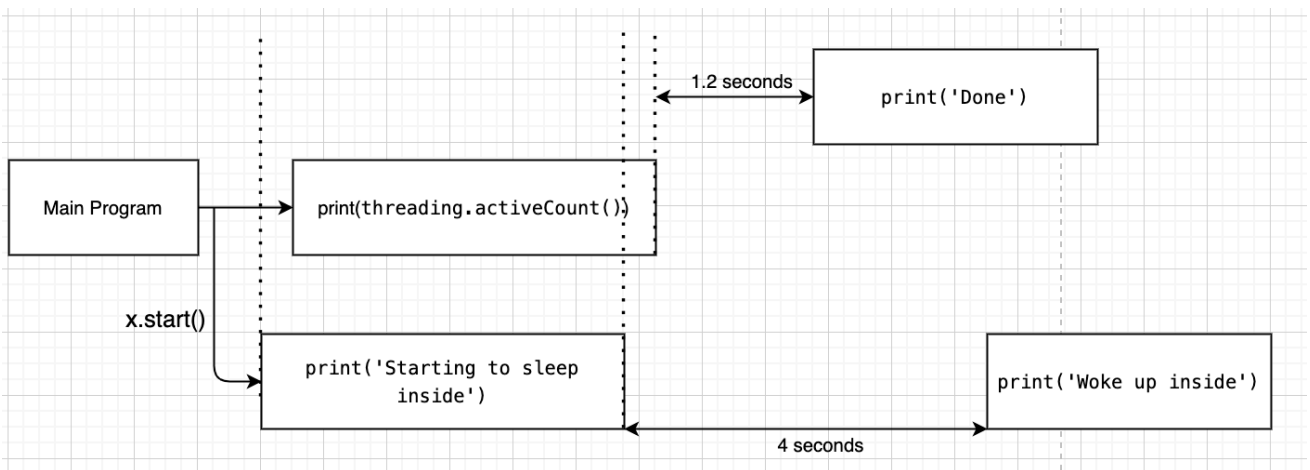
def sleepy_man(secs):
    print(' Starting to sleep inside')
    time.sleep(secs)
    print(' Woke up inside')

x = threading.Thread(target = sleepy_man, args = (4,))
x.start()
print(threading.activeCount())
time.sleep(1.2)
print('Done')
```

The output is as follows-

```
Starting to sleep inside
2
Done
Woke up inside
```

The flow is given in the diagram below-



Now let's spice things a bit. Let us run a for loop that triggers multiple threads.

```
import threading
import time

def sleepy_man(secs):
    print(' Starting to sleep inside - Iteration {}'.format(5-secs))
    time.sleep(secs)
    print(' Woke up inside - Iteration {}'.format(5-secs))

for i in range(3):
    x = threading.Thread(target = sleepy_man, args = (5-i,))
    x.start()

print('Active threads- ', threading.activeCount())
```

At every iteration, we trigger a thread. Note that we pass the arguments 5, 4, 3 at 1st, 2nd, and 3rd iteration respectively. Thus the *sleepy_man()* sleeps 5 seconds, 4 seconds, and 3 seconds respectively.

Thus the output is as shown-

```
Starting to sleep inside - Iteration 0
Starting to sleep inside - Iteration 1
Starting to sleep inside - Iteration 2
```

Active threads- 4

Woke up inside - Iteration 2

Woke up inside - Iteration 1

Woke up inside - Iteration 0

Thus we have seen how multiple threads can be defined and triggered, ensuring a better way of processing which is very essential for heavy I/O operations.

The python programming language allows you to use multiprocessing or multithreading. In this tutorial, you will learn how to write multithreaded applications in Python.

What is a Thread?

A thread is a unit of execution on concurrent programming. Multithreading is a technique which allows a CPU to execute many tasks of one process at the same time. These threads can execute individually while sharing their process resources.

What is a Process?

A process is basically the program in execution. When you start an application in your computer (like a browser or text editor), the operating system creates a **process**.

What is Multithreading in Python?

Multithreading in Python programming is a well-known technique in which multiple threads in a process share their data space with the main thread which makes information sharing and communication within threads easy and efficient. Threads are lighter than processes. Multi threads may execute individually while sharing their process resources. The purpose of multithreading is to run multiple tasks and function calls at the same time.

What is Multiprocessing?

[Multiprocessing](#) allows you to run multiple unrelated processes simultaneously. These processes do not share their resources and communicate through IPC.

Python Multithreading vs Multiprocessing

To understand processes and threads, consider this scenario: An .exe file on your computer is a program. When you open it, the OS loads it into memory, and the CPU executes it. The instance of the program which is now running is called the process.

Every process will have 2 fundamental components:

- The Code
- The Data

Now, a process can contain one or more sub-parts called **threads**. This depends on the OS architecture. You can think about a thread as a section of the process which can be executed separately by the operating system.

In other words, it is a stream of instructions which can be run independently by the OS. Threads within a single process share the data of that process and are designed to work together for facilitating parallelism.

Why use Multithreading?

Multithreading allows you to break down an application into multiple sub-tasks and run these tasks simultaneously. If you use multithreading properly, your application speed, performance, and rendering can all be improved.

Python MultiThreading

[Python](#) supports constructs for both multiprocessing as well as multithreading. In this tutorial, you will primarily be focusing on implementing **multithreaded** applications with python. There are two main modules which can be used to handle threads in Python:

1. The **thread** module, and
2. The **threading** module

However, in python, there is also something called a global interpreter lock (GIL). It doesn't allow for much performance gain and may even **reduce** the performance of some multithreaded applications.

The Thread and Threading modules

The two modules that you will learn about in this tutorial are the **thread module** and the **threading module**.

However, the thread module has long been deprecated. Starting with Python 3, it has been designated as obsolete and is only accessible as **__thread** for backward compatibility.

You should use the higher-level **threading** module for applications which you intend to deploy. The thread module has only been covered here for educational purposes.

The Thread Module

The syntax to create a new thread using this module is as follows:

```
thread.start_new_thread(function_name, arguments)
```

Alright, now you have covered the basic theory to start coding. So, open your IDLE or a notepad and type in the following:

```
import time
import _thread

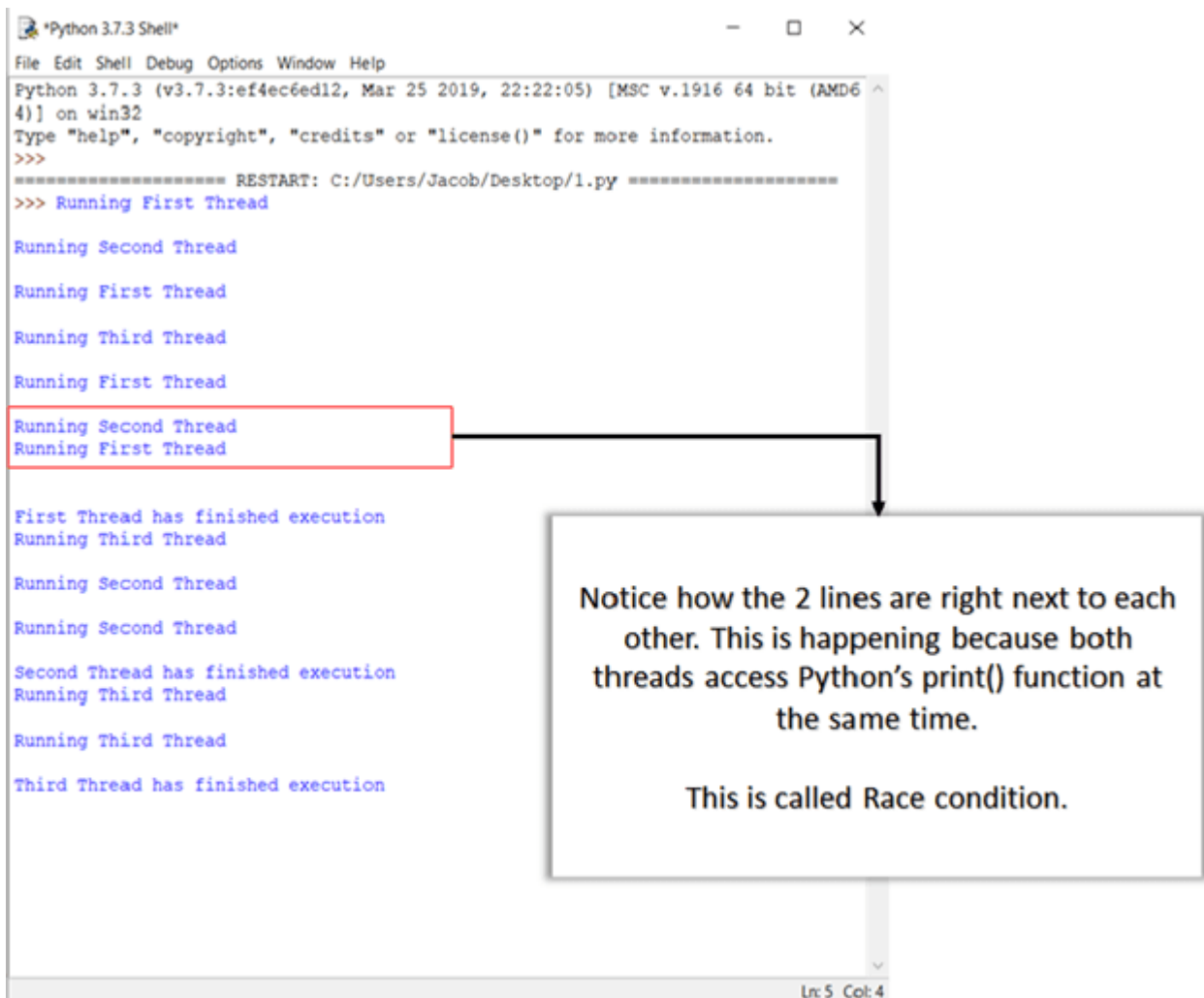
def thread_test(name, wait):
    i = 0
    while i <= 3:
        time.sleep(wait)
        print("Running %s\n" %name)
        i = i + 1

    print("%s has finished execution" %name)

if __name__ == "__main__":

    _thread.start_new_thread(thread_test, ("First Thread", 1))
    _thread.start_new_thread(thread_test, ("Second Thread", 2))
    _thread.start_new_thread(thread_test, ("Third Thread", 3))
```

Save the file and hit F5 to run the program. If everything was done correctly, this is the output that you should see:



The screenshot shows a Python 3.7.3 Shell window with the following output:

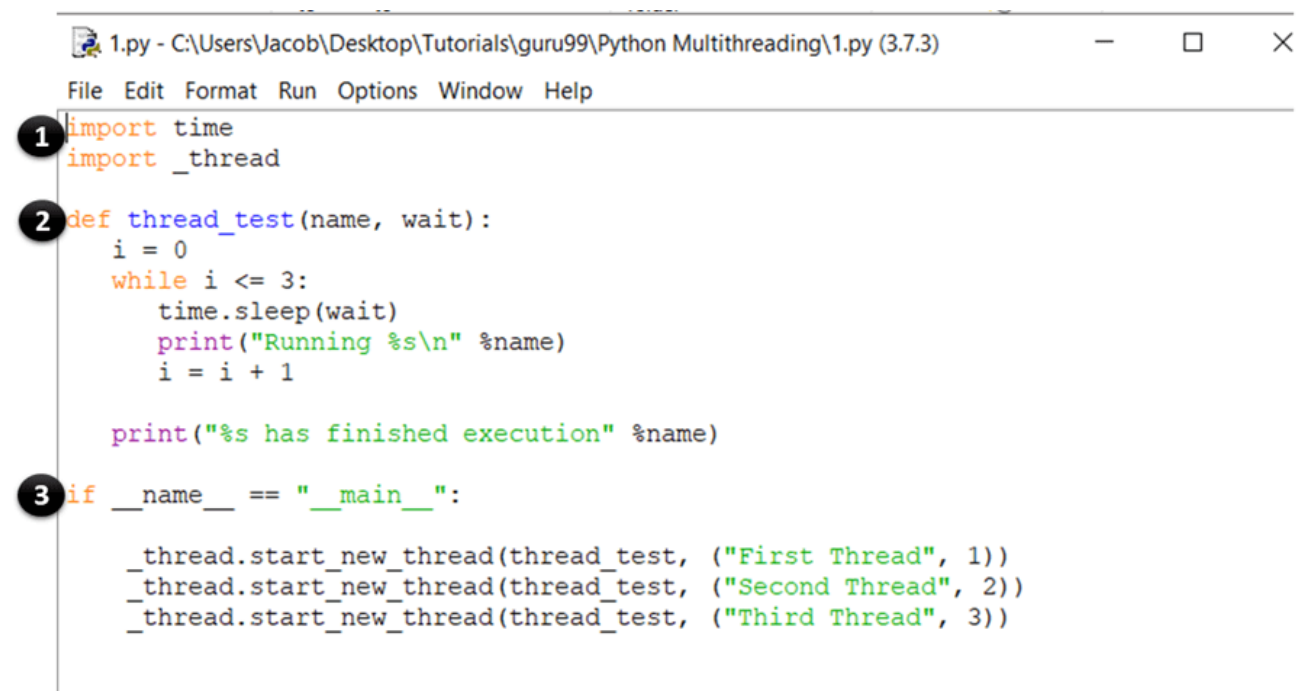
```
Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 22:22:05) [MSC v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: C:/Users/Jacob/Desktop/1.py =====
>>> Running First Thread
Running Second Thread
Running First Thread
Running Third Thread
Running First Thread
Running Second Thread
Running First Thread
First Thread has finished execution
Running Third Thread
Running Second Thread
Running Second Thread
Second Thread has finished execution
Running Third Thread
Running Third Thread
Third Thread has finished execution
```

A red box highlights the lines "Running Second Thread" and "Running First Thread" in the output. An arrow points from this box to a callout box containing the text:

Notice how the 2 lines are right next to each other. This is happening because both threads access Python's print() function at the same time.

This is called Race condition.

You will learn more about race conditions and how to handle them in the upcoming sections



The screenshot shows a Python IDE window titled "1.py - C:/Users/Jacob/Desktop/Tutorials/guru99/Python Multithreading/1.py (3.7.3)". The code is as follows:

```
1 import time
import _thread

2 def thread_test(name, wait):
    i = 0
    while i <= 3:
        time.sleep(wait)
        print("Running %s\n" %name)
        i = i + 1

    print("%s has finished execution" %name)

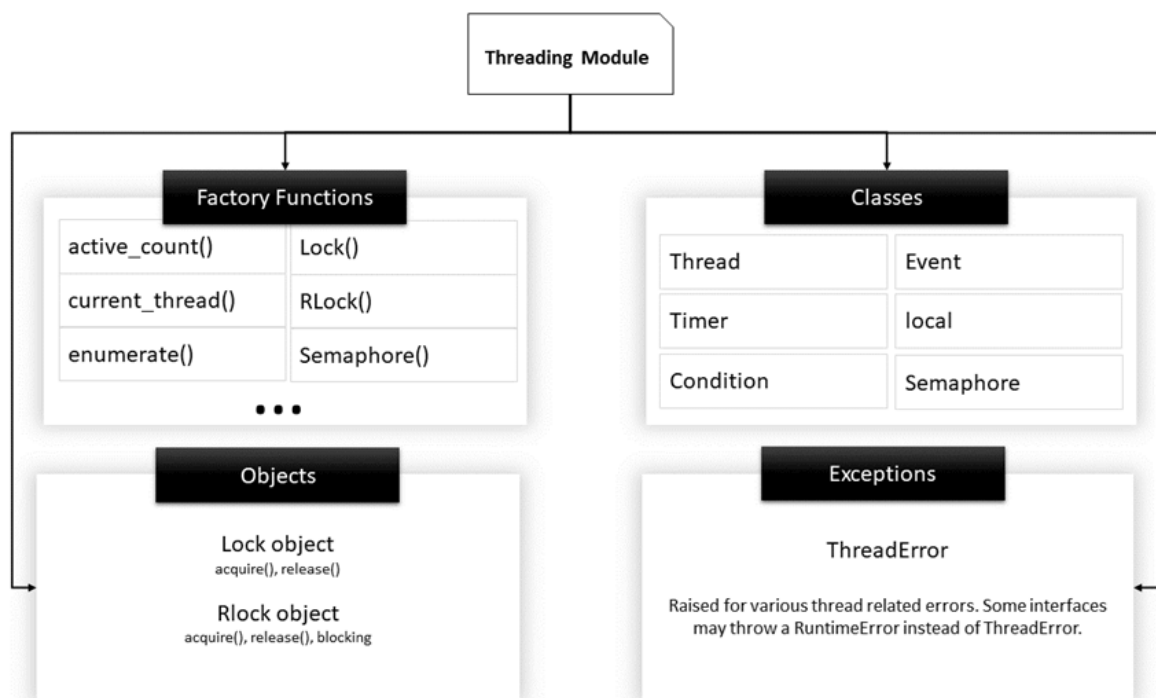
3 if __name__ == "__main__":
    _thread.start_new_thread(thread_test, ("First Thread", 1))
    _thread.start_new_thread(thread_test, ("Second Thread", 2))
    _thread.start_new_thread(thread_test, ("Third Thread", 3))
```

CODE EXPLANATION

1. These statements import the time and thread module which are used to handle the execution and delaying of the Python threads.
2. Here, you have defined a function called **thread_test**, which will be called by the **start_new_thread** method. The function runs a while loop for four iterations and prints the name of the thread which called it. Once the iteration is complete, it prints a message saying that the thread has finished execution.
3. This is the main section of your program. Here, you simply call the **start_new_thread** method with the **thread_test** function as an argument. This will create a new thread for the function you pass as argument and start executing it. Note that you can replace this (thread_test) with any other function which you want to run as a thread.

The Threading Module

This module is the high-level implementation of threading in python and the de facto standard for managing multithreaded applications. It provides a wide range of features when compared to the thread module.



Structure of Threading module

Here is a list of some useful functions defined in this module:

Function Name	Description
activeCount()	Returns the count of Thread objects which are still alive

currentThread() Returns the current object of the Thread class.

enumerate() Lists all active Thread objects.

isDaemon() Returns true if the thread is a daemon.

isAlive() Returns true if the thread is still alive.

Thread Class methods

start() Starts the activity of a thread. It must be called only once for each thread because it will throw a runtime error if called multiple times.

run() This method denotes the activity of a thread and can be overridden by a class that extends the Thread class.

join() It blocks the execution of other code until the thread on which the join() method was called gets terminated.

Backstory: The Thread Class

Before you start coding multithreaded programs using the threading module, it is crucial to understand about the Thread class. The thread class is the primary class which defines the template and the operations of a thread in python.

The most common way to create a multithreaded python application is to declare a class which extends the Thread class and overrides its run() method.

The Thread class, in summary, signifies a code sequence that runs in a separate **thread** of control.

So, when writing a multithreaded app, you will do the following:

1. define a class which extends the Thread class
2. Override the **__init__** constructor
3. Override the **run()** method

Once a thread object has been made, the **start()** method can be used to begin the execution of this activity and the **join()** method can be used to block all other code till the current activity finishes.

Now, let's try using the threading module to implement your previous example. Again, fire up your IDLE and type in the following:

```
import time
import threading
```

```
class threadtester (threading.Thread):
```



```

def __init__(self, id, name, i):
    threading.Thread.__init__(self)
    self.id = id
    self.name = name
    self.i = i

def run(self):
    thread_test(self.name, self.i, 5)
    print ("%s has finished execution " %self.name)

def thread_test(name, wait, i):

    while i:
        time.sleep(wait)
        print ("Running %s \n" %name)
        i = i - 1

if __name__=="__main__":
    thread1 = threadtester(1, "First Thread", 1)
    thread2 = threadtester(2, "Second Thread", 2)
    thread3 = threadtester(3, "Third Thread", 3)

    thread1.start()
    thread2.start()
    thread3.start()

    thread1.join()
    thread2.join()
    thread3.join()

```

This will be the output when you execute the above code:

```
Python 3.7.3 Shell
File Edit Shell Debug Options Window Help
4)] on win32
Type "help", "copyright", "credits" or "license()" for more information.

===== RESTART: C:/Users/Jacob/Desktop/3.py =====
>>>
===== RESTART: C:/Users/Jacob/Desktop/3.py =====
Running First Thread

Running Second Thread

Running First Thread

Running Third Thread

Running First Thread

Running Second Thread

Running First Thread

Running First Thread

First Thread has finished execution
Running Third Thread
Running Second Thread

Running Second Thread

Running Third Thread

Running Second Thread

Second Thread has finished execution
Running Third Thread

Running Third Thread

Third Thread has finished execution
>>>
```

CODE EXPLANATION

```

1 import time
  import threading

  class threadtester (threading.Thread):
2      def __init__(self, id, name, i):
          threading.Thread.__init__(self)
          self.id = id
          self.name = name
          self.i = i

          def run(self):
              thread_test(self.name, self.i, 5)
              print ("%s has finished execution " %self.name)

3 def thread_test(name, delay, i):

    while i:
        time.sleep(delay)
        print ("Running %s \n" %name)
        i = i - 1

    if __name__ == "__main__":
4        thread1 = threadtester(1, "First Thread", 1)
5        thread2 = threadtester(2, "Second Thread", 2)
        thread3 = threadtester(3, "Third Thread", 3)

        thread1.start()
        thread2.start()
        thread3.start()

6        thread1.join()
        thread2.join()
        thread3.join()

```

1. This part is the same as our previous example. Here, you import the time and thread module which are used to handle the execution and delays of the Python threads.
2. In this bit, you are creating a class called threadtester, which inherits or extends the **Thread** class of the threading module. This is one of the most common ways of creating threads in python. However, you should only override the constructor and the **run()** method in your app. As you can see in the above code sample, the **__init__** method (constructor) has been overridden. Similarly, you have also overridden the **run()** method. It contains the code that you want to execute inside a thread. In this example, you have called the **thread_test()** function.
3. This is the **thread_test()** method which takes the value of **i** as an argument, decreases it by 1 at each iteration and loops through the rest of the code until **i** becomes 0. In each iteration, it prints the name of the currently executing thread and sleeps for wait seconds (which is also taken as an argument).

4. `thread1 = threadtester(1, "First Thread", 1)` Here, we are creating a thread and passing the three parameters that we declared in `__init__`. The first parameter is the id of the thread, the second parameter is the thread's name, and the third parameter is the counter, which determines how many times the while loop should run.
5. `thread2.start()` The start method is used to start the execution of a thread. Internally, the `start()` function calls the `run()` method of your class.
6. `thread3.join()` The `join()` method blocks the execution of other code and waits until the thread on which it was called finishes.

As you already know, the threads which are in the same process have access to the memory and data of that process. As a result, if more than one thread tries to change or access the data simultaneously, errors may creep in.

In the next section, you will see the different kinds of complications that can show up when threads access data and critical-section without checking for existing access transactions.

Deadlocks and Race conditions

Before learning about deadlocks and race conditions, it'll be helpful to understand a few basic definitions related to concurrent programming:

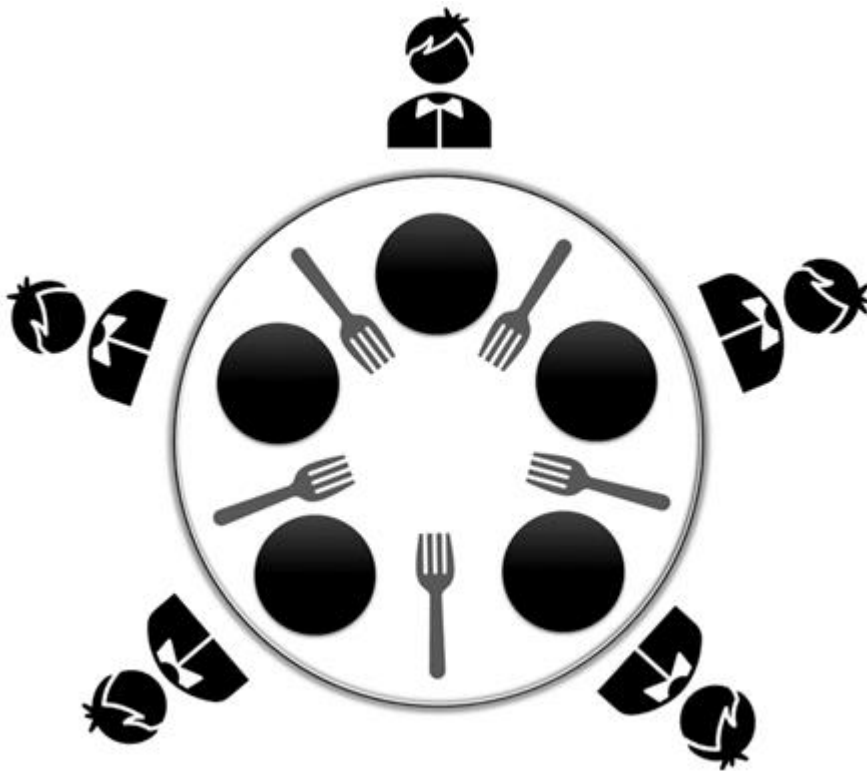
- **Critical Section** It is a fragment of code that accesses or modifies shared variables and must be performed as an atomic transaction.
- **Context Switch** It is the process that a CPU follows to store the state of a thread before changing from one task to another so that it can be resumed from the same point later.

Deadlocks

[Deadlocks](#) are the most feared issue that developers face when writing concurrent/multithreaded applications in python. The best way to understand deadlocks is by using the classic computer science example problem known as the **Dining Philosophers Problem**.

The problem statement for dining philosophers is as follows:

Five philosophers are seated on a round table with five plates of spaghetti (a type of pasta) and five forks, as shown in the diagram.



Dining

Philosophers

Problem

At any given time, a philosopher must either be eating or thinking.

Moreover, a philosopher must take the two forks adjacent to him (i.e., the left and right forks) before he can eat the spaghetti. The problem of deadlock occurs when all five philosophers pick up their right forks simultaneously.

Since each of the philosophers has one fork, they will all wait for the others to put their fork down. As a result, none of them will be able to eat spaghetti.

Similarly, in a concurrent system, a deadlock occurs when different threads or processes (philosophers) try to acquire the shared system resources (forks) at the same time. As a result, none of the processes get a chance to execute as they are waiting for another resource held by some other process.

Race Conditions

A race condition is an unwanted state of a program which occurs when a system performs two or more operations simultaneously. For example, consider this simple for loop:

```
i=0; # a global variable
for x in range(100):
    print(i)
    i+=1;
```

If you create n number of threads which run this code at once, you cannot determine the value of i (which is shared by the threads) when the program

finishes execution. This is because in a real multithreading environment, the threads can overlap, and the value of `i` which was retrieved and modified by a thread can change in between when some other thread accesses it.

These are the two main classes of problems that can occur in a multithreaded or distributed python application. In the next section, you will learn how to overcome this problem by synchronizing threads.

Synchronizing threads

To deal with race conditions, deadlocks, and other thread-based issues, the threading module provides the **Lock** object. The idea is that when a thread wants access to a specific resource, it acquires a lock for that resource. Once a thread locks a particular resource, no other thread can access it until the lock is released. As a result, the changes to the resource will be atomic, and race conditions will be averted.

A lock is a low-level synchronization primitive implemented by the `__thread` module. At any given time, a lock can be in one of 2 states: **locked** or **unlocked**. It supports two methods:

1. **acquire()** When the lock-state is unlocked, calling the `acquire()` method will change the state to locked and return. However, If the state is locked, the call to `acquire()` is blocked until the `release()` method is called by some other thread.
2. **release()** The `release()` method is used to set the state to unlocked, i.e., to release a lock. It can be called by any thread, not necessarily the one that acquired the lock.

Here's an example of using locks in your apps. Fire up your IDLE and type the following:

```
import threading
lock = threading.Lock()

def first_function():
    for i in range(5):
        lock.acquire()
        print('lock acquired')
        print('Executing the first function')
        lock.release()

def second_function():
    for i in range(5):
        lock.acquire()
```

```

    print('lock acquired')
    print('Executing the second funcion')
    lock.release()


if __name__=="__main__":
    thread_one = threading.Thread(target=first_function)
    thread_two = threading.Thread(target=second_function)

    thread_one.start()
    thread_two.start()

    thread_one.join()
    thread_two.join()

```

Now, hit F5. You should see an output like this:



```

Python 3.7.3 Shell
File Edit Shell Debug Options Window Help
Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 22:22:05) [MSC v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: C:/Users/Jacob/Desktop/locks.py =====
lock acquired
Executing the first function
lock acquired
Executing the first function
lock acquired
Executing the first function
lock acquired
Executing the first function
lock acquired
Executing the first function
lock acquired
Executing the second function
lock acquired
Executing the second function
lock acquired
Executing the second function
lock acquired
Executing the second function
lock acquired
Executing the second function
lock acquired
Executing the second function
>>>

```

CODE EXPLANATION

1

```
import threading
```

```
lock = threading.Lock()
```

```
def first_function():
```

```
    for i in range(5):
```

```
        lock.acquire()
```

```
        print ('lock acquired')
```

```
        print ('Executing the first funcion')
```

```
        lock.release()
```

```
def second_function():
```

```
    for i in range(5):
```

```
        lock.acquire()
```

```
        print ('lock acquired')
```

```
        print ('Executing the second funcion')
```

```
        lock.release()
```

```
if __name__=="__main__":
```

```
    thread_one = threading.Thread(target=first_function)
```

```
    thread_two = threading.Thread(target=second_function)
```

```
    thread_one.start()
```

```
    thread_two.start()
```

```
    thread_one.join()
```

```
    thread_two.join()
```

1. Here, you are simply creating a new lock by calling the **threading.Lock()** factory function. Internally, **Lock()** returns an instance of the most effective concrete **Lock** class that is maintained by the platform.
2. In the first statement, you acquire the lock by calling the **acquire()** method. When the lock has been granted, you print **"lock acquired"** to the console. Once all the code that you want the thread to run has finished execution, you release the lock by calling the **release()** method.

The theory is fine, but how do you know that the lock really worked? If you look at the output, you will see that each of the print statements is printing exactly one line at a time. Recall that, in an earlier example, the outputs from print were haphazard because multiple threads were accessing the **print()** method at the same time. Here, the print function is called only after the lock is acquired. So, the outputs are displayed one at a time and line by line.

Apart from locks, python also supports some other mechanisms to handle thread synchronization as listed below:

1. RLocks
2. Semaphores
3. Conditions
4. Events, and
5. Barriers

Global Interpreter Lock (and how to deal with it)

Before getting into the details of python's GIL, let's define a few terms that will be useful in understanding the upcoming section:

1. CPU-bound code: this refers to any piece of code which will be directly executed by the CPU.
2. I/O-bound code: this can be any code that accesses the file system thru' the OS
3. CPython: it is the reference **implementation** of Python and can be described as the interpreter written in C and Python (programming language).

What is GIL in Python?

Global Interpreter Lock (GIL) in python is a process lock or a mutex used while dealing with the processes. It makes sure that one thread can access a particular resource at a time and it also prevents the use of objects and bytecodes at once. This benefits the single-threaded programs in a performance increase. GIL in python is very simple and easy to implement. A lock can be used to make sure that only one thread has access to a particular resource at a given time.

One of the features of Python is that it uses a global lock on each interpreter process, which means that every process treats the python interpreter itself as a resource.

For example, suppose you have written a python program which uses two threads to perform both CPU and 'I/O' operations. When you execute this program, this is what happens:

1. The python interpreter creates a new process and spawns the threads
2. When thread-1 starts running, it will first acquire the GIL and lock it.

3. If thread-2 wants to execute now, it will have to wait for the GIL to be released even if another processor is free.
4. Now, suppose thread-1 is waiting for an I/O operation. At this time, it will release the GIL, and thread-2 will acquire it.
5. After completing the I/O ops, if thread-1 wants to execute now, it will again have to wait for the GIL to be released by thread-2.

Due to this, only one thread can access the interpreter at any time, meaning that there will be only one thread executing python code at a given point of time.

This is alright in a single-core processor because it would be using time slicing (see the first section of this tutorial) to handle the threads. However, in case of multi-core processors, a CPU-bound function executing on multiple threads will have a considerable impact on the program's efficiency since it won't actually be using all the available cores at the same time.

Why was GIL needed?

The CPython garbage collector uses an efficient memory management technique known as reference counting. Here's how it works: Every object in python has a reference count, which is increased when it is assigned to a new variable name or added to a container (like tuples, lists, etc.). Likewise, the reference count is decreased when the reference goes out of scope or when the del statement is called. When the reference count of an object reaches 0, it is garbage collected, and the allotted memory is freed.

But the problem is that the reference count variable is prone to race conditions like any other global variable. To solve this problem, the developers of python decided to use the global interpreter lock. The other option was to add a lock to each object which would have resulted in deadlocks and increased overhead from acquire() and release() calls.

Therefore, GIL is a significant restriction for multithreaded python programs running heavy CPU-bound operations (effectively making them single-threaded). If you want to make use of multiple CPU cores in your application, use the **multiprocessing** module instead.

Summary

- Python supports 2 modules for multithreading:

1. **__thread** module: It provides a low-level implementation for threading and is obsolete.
 2. **threading module**: It provides a high-level implementation for multithreading and is the current standard.
- To create a thread using the threading module, you must do the following:
 1. Create a class which extends the **Thread** class.
 2. Override its constructor (**__init__**).
 3. Override its **run()** method.
 4. Create an object of this class.
 - A thread can be executed by calling the **start()** method.
 - The **join()** method can be used to block other threads until this thread (the one on which join was called) finishes execution.
 - A race condition occurs when multiple threads access or modify a shared resource at the same time.
 - It can be avoided by Synchronizing threads.
 - Python supports 6 ways to synchronize threads:
 1. Locks
 2. RLocks
 3. Semaphores
 4. Conditions
 5. Events, and
 6. Barriers
 - Locks allow only a particular thread which has acquired the lock to enter the critical section.
 - A Lock has 2 primary methods:
 1. **acquire()**: It sets the lock state to **locked**. If called on a locked object, it blocks until the resource is free.
 2. **release()**: It sets the lock state to **unlocked** and returns. If called on an unlocked object, it returns false.
 - The global interpreter lock is a mechanism through which only 1 CPython interpreter process can execute at a time.
 - It was used to facilitate the reference counting functionality of CPython's garbage collector.
 - To make Python apps with heavy CPU-bound operations, you should use the multiprocessing module.