Multithreaded Programming

A **thread** is an entity within a process that can be scheduled for execution. Also, it is the smallest unit of processing that can be performed in an OS (Operating System).

In simple words, a **thread** is a sequence of such instructions within a program that can be executed independently of other code. For simplicity, you can assume that a thread is simply a subset of a process!

- For a single process, multiple threads can be used to process and share the same data-space and can communicate with each other by sharing information.
- They use lesser memory overhead, and hence they are called lightweight processes.
- A program can remain responsive to input when threads are used.
- Threads can share and process the global variable's memory.

In a thread, there are three different parts. It has the beginning, an execution part, and a conclusion. It also has an instruction pointer that points to where the thread or process is currently running, and hence the thread can run several different program blocks concurrently.

In Python, the **threading** module provides a very simple and intuitive API for spawning multiple threads in a program.

Python Multithreading Modules

Python offers two modules to implement threads in programs.

- <thread> module
- <threading> module

The key difference between the two modules is that the module <thread> implements a thread as a function. On the other hand, the module <threading> offers an object-oriented approach to enable thread creation.

Thread Module

the <thread> module to apply in your program, then use the following method to spawn threads.

```
thread.start_new_thread(function, args[, kwargs])
```

Here, the first part is a method as told before & this method is a faster and more efficient way to create new threads. As the child thread starts the function passes a list of args. The thread gets terminated when the function returns a value. The 'args' in the above syntax is a tuple of arguments.

Example	Output
from thread import start_new_thread def add(a,b): print a+b start_new_thread(add,(10,20,))	Waiting for the thread. 30 50

start_new_thread(add,(20,30,)) c = raw_input("Waiting for the thread.\n")	

The Threading Module

The latest <threading> module provides rich features and greater support for threads than the legacy <thread> module. The <threading> module is an excellent example of Python Multithreading.

The latest <threading> module provides rich features and greater support for threads than the legacy <thread> module. The <threading> module is an excellent example of Python Multithreading.

The <threading> module combines all the methods of the <thread> module and exposes few additional methods.

- threading.activeCount(): It finds the total no. of active thread objects.
- threading.currentThread(): You can use it to determine the number of thread objects in the caller's thread control.
- threading.enumerate(): It will give you a complete list of thread objects that are currently active.

Apart from the above methods, <threading> module also presents the <Thread> class that you can try for implementing threads. It is an object-oriented variant of Python multithreading.

The <Thread> class publishes the following methods

- run(): The run() method is the entry point for a thread.
- start(): The start() method starts a thread by calling the run method.
- join([time]): The join() waits for threads to terminate.
- isAlive(): The isAlive() method checks whether a thread is still executing.
- getName(): The getName() method returns the name of a thread.
- setName(): The setName() method sets the name of a thread.

Example	Output
import threading def add(a,b): print a+b t1 = threading.Thread(target=add, args=(10,20,)) t2 = threading.Thread(target=add, args=(20,30,)) t1.start() t2.start() t1.join() t2.join()	30 50