Python Multithreading

Multithreading is a core concept of software programming that almost all the high-level programming languages support. And Python multithreading is one of the best examples of simplistic implementation of threads.

In software programming, a thread is the smallest unit of execution with the independent set of instructions. It is a part of the process and operates in the same context sharing program's runnable resources like memory. A thread has a starting point, an execution sequence, and a result.

The ability of a process to execute multiple threads parallelly is called multithreading. Ideally, multithreading can significantly improve the performance of any program.

Python Multithreading – Pros:

- Multithreading can significantly improve the speed of computation on multiprocessor or multi-core systems because each processor or core handles a separate thread concurrently.
- Multithreading allows a program to remain responsive while one thread waits for input and another runs a GUI at the same time. This statement holds true for both multiprocessor or single processor systems.
- All the threads of a process have access to its global variables. If a global variable changes in one thread, it is visible to other threads as well. A thread can also have its own local variables.

Python Multithreading – Cons:

- On a single processor system, multithreading wouldn't impact the speed of computation. In fact, the system's performance may downgrade due to the overhead of managing threads.
- Synchronization is required to avoid mutual exclusion while accessing shared resources of the process. It directly leads to more memory and CPU utilization.
- Multithreading increases the complexity of the program thus also making it difficult to debug.
- It raises the possibility of potential deadlocks.
- It may cause starvation when a thread doesn't get regular access to shared resources. It would then fail to resume its work.

Python Multithreading Modules For Thread Implementation.

Python offers two modules to implement threads in programs.

- <thread> module and
- <threading> module.

<thread> module is deprecated in Python 3 and renamed to <_thread> module for backward compatibility. 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.

How To Use The Threading Module To Create Threads

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.

Class Methods	Method Description
run():	It is the entry point function for any thread.
start():	The start() method triggers a thread when run method is called.
join([time]):	The join() method enables a program to wait for threads to terminate.
isAlive():	The isAlive() method verifies an active thread.
getName():	The getName() method retrieves the name of a thread.
setName():	The setName() method updates the name of a thread.

Steps To Implement Threads Using The Threading Module.

You may follow the below steps to implement a new thread using the <threading> module.

- Construct a subclass from the <Thread> class.
- Override the <__init__(self [,args])> method to supply arguments as per requirements.
- Next, override the <run(self [,args])> method to code the business logic of the thread.

Once you define the new <Thread> subclass, you have to instantiate it to start a new thread. Then, invoke the <start()> method to initiate it. It will eventually call the <run()> method to execute the business logic.

Example – Create A Thread Class And Object To Print Current Date.

```
#Python multithreading example to print current date.
#1. Define a subclass using Thread class.
#2. Instantiate the subclass and trigger the thread.

import threading
import datetime

class myThread (threading.Thread):
    def __init__(self, name, counter):
        threading.Thread.__init__(self)
        self.threadID = counter
        self.name = name
        self.counter = counter
    def run(self):
        print "Starting " + self.name
        print_date(self.name, self.counter)
        print "Exiting " + self.name

def print_date(threadName, counter):
    datefields = []
    today = datetime.date.today()
    datefields.append(today)
    print ("%s[%d]: %s") % ( threadName, counter,
datefields[0] )
```

```
# Create new threads
thread1 = myThread("Thread", 1)
thread2 = myThread("Thread", 2)

# Start new Threads
thread1.start()
thread2.start()

thread1.join()
thread2.join()
print "Exiting the Program!!!"
```