# Day 28 – Python - Multithreaded Programming

* Multiple threads within a process share the same data space with the main thread and can therefore share information or communicate with each other more easily than if they were separate processes.
* Threads sometimes called light-weight processes and they do not require much memory overhead; they are cheaper than processes.
* A thread has a beginning, an execution sequence, and a conclusion. It has an instruction pointer that keeps track of where within its context it is currently running.
* It can be pre-empted (interrupted)
* It can temporarily be put on hold (also known as sleeping) while other threads are running - this is called yielding.
* To spawn another thread, you need to call following method available in *thread* module –

thread.start\_new\_thread ( function, args[, kwargs] )

#!/usr/bin/python

import thread

import time

# Define a function for the thread

def print\_time( threadName, delay):

count = 0

while count < 5:

time.sleep(delay)

count += 1

print "%s: %s" % ( threadName, time.ctime(time.time()) )

# Create two threads as follows

try:

thread.start\_new\_thread( print\_time, ("Thread-1", 2, ) )

thread.start\_new\_thread( print\_time, ("Thread-2", 4, ) )

except:

print "Error: unable to start thread"

while 1:

pass

* The newer threading module included with Python 2.4 provides much more powerful, high-level support for threads than the thread module discussed in the previous section.
* The *threading* module exposes all the methods of the *thread* module and provides some additional methods −
* **threading.activeCount()** − Returns the number of thread objects that are active.
* **threading.currentThread()** − Returns the number of thread objects in the caller's thread control.
* **threading.enumerate()** − Returns a list of all thread objects that are currently active.

In addition to the methods, the threading module has the *Thread* class that implements threading. The methods provided by the *Thread* class are as follows −

* **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.

To implement a new thread using the threading module, you have to do the following −

* Define a new subclass of the *Thread* class.
* Override the *\_\_init\_\_(self [,args])* method to add additional arguments.
* Then, override the run(self [,args]) method to implement what the thread should do when started.
* Once you have created the new *Thread* subclass, you can create an instance of it and then start a new thread by invoking the *start()*, which in turn calls *run()* method.
* The threading module provided with Python includes a simple-to-implement locking mechanism that allows you to synchronize threads. A new lock is created by calling the *Lock()* method, which returns the new lock.

The *Queue* module allows you to create a new queue object that can hold a specific number of items. There are following methods to control the Queue −

* **get()** − The get() removes and returns an item from the queue.
* **put()** − The put adds item to a queue.
* **qsize()** − The qsize() returns the number of items that are currently in the queue.
* **empty()** − The empty( ) returns True if queue is empty; otherwise, False.
* **full()** − the full() returns True if queue is full; otherwise, False.

Exercise:

Write a Python multithreading program to print current date.

#1. Define a subclass using threading.Thread class.

#2. Instantiate the subclass and trigger the thread.