PYTHON OOPs

Multithreading :

- Multithreading is a core concept of software programming that almost all the high-level programming languages support.
- The ability of a process to execute multiple threads parallely is called multithreading.
- Ideally, multithreading can significantly improve the performance of any program.
- we'll learn different methods to create threads and implement synchronization.

Multithreading Modules For Thread Implementation **:**

- Python offers two modules to implement threads in programs.

<thread> module.

<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 Thread Module To Create Threads :

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

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

- This method is quite simple and efficient way to create threads. You can use it to run programs in both Linux and Windows.

- This method starts a new thread and returns its identifier. It’ll invoke the function specified as the “function” parameter with the passed list of arguments. When the <function> returns, the thread would silently exit.

- Here, args is a tuple of arguments; use an empty tuple to call <function> without any arguments. The optional <kwargs> argument specifies the dictionary of keyword arguments.

- If the <function> terminates with an unhandled exception, a stack trace is printed and then the thread exits (It doesn’t affect other threads, they continue to run). Use the below code to learn more about threading.

Example :

```
from _thread  import start_new_thread

threadId = 1

def factorial(n):

    global threadId

    if n < 1:

         print ("%s: %d" % ("Thread",threadId ))

        threadId = threadId + 1

        return 1

    else:

        returnNumber = n * factorial( n-1 )

       print(str(n) + '! = ' + str(returnNumber))

        return returnNumber

start_new_thread(factorial,(5, ))
start_new_thread(factorial,(4, ))

c = input("Waiting for threads to return...\n")
```

Output :

```
 Waiting for threads to return...

 Thread: 1

 Thread: 1

 1! = 1

2! = 2

3! = 6

4! = 24

1! = 1

5! = 120

2! = 2

3! = 6

4! = 24
```

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.

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

Methods in <thread> module :

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.

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 :

```
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]))

thread1 = myThread("Thread", 1)

thread2 = myThread("Thread", 2)

thread1.start()

thread2.start()

thread1.join()

thread2.join()
```

Output :

```
Starting Thread

Thread[1]: 2017-10-15

Starting Thread

Exiting Thread

Thread[2]: 2017-10-15

Exiting Thread

Exiting the Program!!!
```

Synchronizing Thread :

· The <*threading*> module has built in functionality to implement locking that allows you to synchronize threads. Locking is required to control access to shared resources to prevent corruption or missed data.

· You can call *Lock()* method to apply locks, it returns the new lock object. Then, you can invoke the *acquire(blocking)* method of the lock object to enforce threads to run synchronously.

· The optional *blocking* parameter specifies whether the thread waits to acquire the lock.

· In case, *blocking* is set to zero, the thread returns immediately with a zero value if the lock can't be acquired and with a 1 if the lock was acquired.

· In case, *blocking* is set to 1, the thread blocks and wait for the lock to be released.

· The *release()* method of the lock object is used to release the lock when it is no longer required.

Example :

```
import threading

import datetime
```

```
exitFlag = 0

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)

        threadLock.acquire()

        print_date(self.name, self.counter)

        threadLock.release()

        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]))

threadLock = threading.Lock()

threads = []

thread1 = myThread("Thread", 1)

thread2 = myThread("Thread", 2)

thread1.start()

thread2.start()

threads.append(thread1)
threads.append(thread2)

for t in threads:

   a = t.join()

   print (a)

print ("Exiting the Program!!!")
```

Output :

```
 Starting Thread

Thread[1]: 2017-10-15

Starting Thread

Exiting Thread

Thread[2]: 2017-10-15

None

Exiting Thread

None

Exiting the Program!!!
```