Multithreading in Python

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.

Multithreading is defined as the ability of a processor to execute multiple threads concurrently.

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

Example 1:

```
# Python program to illustrate the concept default thread
# of threading
# importing the threading module

import threading
print("Defalut Thread is: ", threading.current_thread().getName())
```

Example 2:

```
'''
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.
'''
import threading

print(threading.activeCount())
print(threading.currentThread())
print(threading.enumerate())
```

1. Every python program has a default thread in it with the name ‘MainThread’.
2. This thread is responsible for execution the program line by line.
3. Instead, user can create his own thread using ‘threading’ module.
4. When another thread is present in the program along with the ‘MainThread’, then both thread executions become independent.
5. We can even set a name for the user defined thread using ‘**setName()**’ function.
6. In order to run the user defined thread we use the function ‘**start()**’

Few Methods of ‘**threading’** module:

- **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 3:

```
# Defining a user-defined thread
import threading

t1 = threading.Thread()
print("Default name of a new thread: ", t1.getName())
t1.setName("MyThread1") #setting new name
print("User Defined name of the new thread: ", t1.getName())
```

Example :

```
import threading

def firstfun():
    for i in range(1,11):
        print("From fun add: " , i)
        pass
    pass

def secondfun():
    for i in range(11,21):
        print("From fun add: " , i)
        pass
    pass


t1 = threading.Thread(target=firstfun)
t2 = threading.Thread(target=secondfun)

t1.start()
t2.start()
```

Example 4:

```
# Multi-tasking example using 'Thread' class of 'threading' module
from threading import *

def display():
    for i in range(10):
        print('Child Thread')
# end of function

#creating a thread object and calling display function
t = Thread(target=display)
t.start()


for i in range(10):
    print('-- Parent Thread')
#calling parent thread


#In the above program child thread is calling 'display()'
#Main thread is executing total program lines as well as child thread
#Here ‘Thread’ is a pre-defined class present in ‘threading’ module.
```

Example 5:

```
# Threading with parameters
from threading import *

def display(start, end):
    for i in range(start,end):
        print('Child Thread')
# end of function
```

```
#creating a thread object and calling display function
t = Thread(target=display,args=(1,11),)
t.start()


for i in range(10):
    print('-- Parent Thread')
```

NOTE: in thread programming the output of the program is not similar all the time. There is no guarantee that a thread program will yield same output all the time.

NOTE: If two or more jobs are independent then go for multi-threading concept.

NOTE: Though 'Main' thread is responsible for calling 'child' thread, once the child thread execution begins then, 'Main' and 'Child' thread executions are separated.

NOTE: A single thread can be executed for only once. Hence we cannot use 'start()' method in a loop.

Example 6:

```
# Python program to illustrate the concept process id of each thread

import threading
import os

def task1():
    print("Task 1 assigned to thread:
{}".format(threading.current_thread().name))
    print("ID of process running task 1: {}\n".format(os.getpid()))

def task2():
    print("Task 2 assigned to thread:
{}".format(threading.current_thread().name))
    print("ID of process running task 2: {}\n".format(os.getpid()))

if __name__ == "__main__":

    # print ID of current process
    print("ID of process running main program: {}".format(os.getpid()))
    # print name of main thread
    print("Main thread name: {}\n".format(threading.current_thread().name))

    # creating threads
    t1 = threading.Thread(target=task1, name='t1')
    t2 = threading.Thread(target=task2, name='t2')

    # starting threads
    t1.start()
    t2.start()
```

Example 7:

```
# Threads in OOP
from threading import *

class Test:
    def display(self):
        for i in range(10):
            print("Child Thread-2")
            pass
        pass
    pass

obj = Test()
t = Thread(target=obj.display)
t.start()

# from main area - main thread
for i in range(10):
    print("Main Thread-1")
    pass
```

Example 8:

```
# Single method executed by multiple threads
from threading import *

class Test:

    def display(self):
        for i in range(5):
            print('Child Thread Executed by: ', current_thread().getName())
            pass
        pass
    pass

obj = Test()
t1 = Thread(target=obj.display)
t2 = Thread(target=obj.display)
t3 = Thread(target=obj.display)
t4 = Thread(target=obj.display)
t5 = Thread(target=obj.display)

t1.start()
t2.start()
t3.start()
t4.start()
t5.start()
```

Example 9:

```
# slow motions in thread execution using 'sleep()' function

import threading
import time
import datetime

def showtime():
    for i in range(1,6):
        print("Thread name is:", threading.current_thread().getName(), end=' ')
        print("Date and Time is:",datetime.datetime.now())
        time.sleep(1)

t1 = threading.Thread(target=showtime,)
t2 = threading.Thread(target=showtime,)

t1.start()
t2.start()
```

Example 10:

```
# Thread use-case
# Threading can save execution time and enhance speed of execution
```

```python
# For this threads uses 'time-slicing'

# Prog without threading:

import time

def doubles(num):
    for n in num:
        time.sleep(1)
        print('Double value: ', 2*n)
        pass

def squares(num):
    for n in num:
        time.sleep(1)
        print('Square value: ', n*n)
        pass
    pass

num = [1,2,3,4,5,6]

start_time = time.time()
doubles(num)
squares(num)
end_time = time.time()

print('The total time taken: ', end_time - start_time)
```

Example 11:

```python
# Prog with threading and join function:

# join() : is a method used to make the main thread or any other thread wait
# until the previous threads completed its execution.

# execute the below prog with and without join() method.
import time
from threading import *

def doubles(num):
    for n in num:
        time.sleep(1)
        print('Double value: ', 2*n)
        pass
```

```
def squares(num):
    for n in num:
        time.sleep(1)
        print('Square value: ', n*n)
        pass
    pass

num = [1,2,3,4,5,6]

start_time = time.time()
#doubles(num)
t1 = Thread(target=doubles, args=(num,))

#squares(num)
t2 = Thread(target=squares, args=(num,))

t1.start()
t2.start()

t1.join()
t2.join()
end_time = time.time()

print('The total time taken: ', end_time - start_time)
```

Synchronization in Python

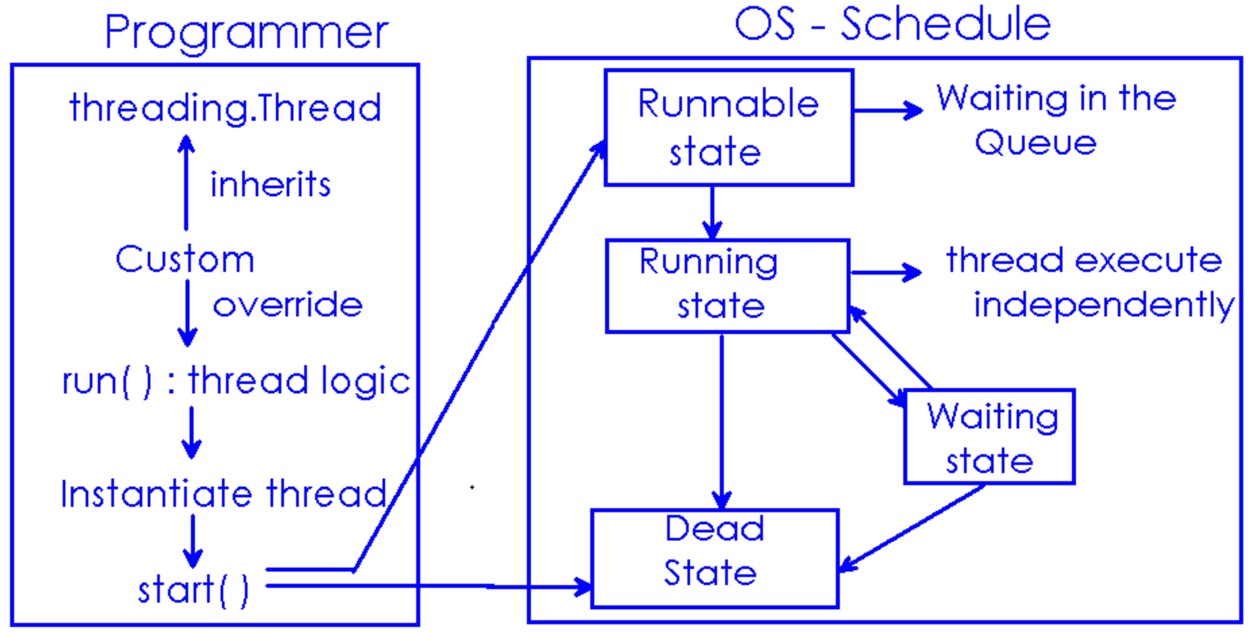
Multithreading allows your computer to perform actions in parallel, utilizing multiple cores/ multiple CPUs present on your system. However, when it comes to reading and updating shared variables at the same time, it can lead to erroneous results.

```
import threading
import time

class myThread (threading.Thread):
   def __init__(self, threadID, name, counter):
      threading.Thread.__init__(self) #super class constr call from sub class
constr
      self.threadID = threadID
      self.name = name
      self.counter = counter

   def run(self):
      print ("Starting " + self.name)
      # Get lock to synchronize threads
      threadLock.acquire() #sync starts
      print("Is lock acquired: ", threadLock.locked())
      print_time(self.name, self.counter, 3)
      # Free lock to release next thread
      threadLock.release() #sync ends
      print("Is still lock acquired: ", threadLock.locked())

def print_time(threadName, delay, c):
   while c:
      time.sleep(delay)
      print ("%s: %s" % (threadName, time.ctime(time.time())))
      c -= 1

# preparing Lock() object for synchronization
threadLock = threading.Lock()
t = []

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

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

# Add threads to thread list
t.append(thread1)
t.append(thread2)

# Wait for all threads to complete
for i in t:
   i.join()
print ("Exiting Main Thread")
```

………………………………………………………………………………………………

Life cycle of Thread:

- Every thread executes according to Life cycle.
- Life cycle consists
    - Creation phase by programmer
    - Execution phase by scheduler(OS)
- As a programmer, we need to define and start the thread.
- All threads scheduled by the processor.
- Thread has many states
    - Runnable state
    - Running state
    - Waiting state
    - Dead state.
- Every thread should wait in the Queue until processor allocates the memory is called Runnable state.
- Once the memory has been allocated to thread, it moves to running state. In running state, thread execute independently.
- From the running state, the thread may fall into waiting state to read user input, sleep and join methods.
- Once the thread execution completed, it moves to dead state.



Python Daemon Threads

The threads which are always going to run in the background that provides supports to main or non-daemon threads, those background executing threads are considered as **Daemon Threads.** The **Daemon Thread** does not block the main thread from exiting and continues to run in the background.

This example simplifies the flow of a non-daemon thread where we have created a **thread_1()** name function which having some lines of instructions to execute which reveal how the non-daemon thread is executed when the main thread terminates. At the next we have created the thread **T** of function **thread_1()** which is currently considered as a non-active thread, now we start the thread T, and we also have temporarily stopped the execution of the main thread for 5secs. Of time, between this 5sec. Thread **T** continues its execution and when the main thread is going to be executed after 5sec. Where it stops its work but there is a thread **T** still is in execution because it is a non-daemon thread and executes their instruction until their completion.

Example for Daemon Thread: Garbage Collector of Python or Java

Example 1:

```
# non-daemon thread behaviour

from threading import *
import time

# creating a function
def thread_1():
    for i in range(5):
        print('this is non-daemon thread')
        time.sleep(2)

# creating a thread T
T = Thread(target=thread_1)

# starting of thread T
T.start()

# main thread stop execution till 5 sec.
```

```
time.sleep(5)
print('main Thread execution')
```

Example 2:

This is an example to show how daemon threads behave over non-daemon threads during program execution. We already see in the above example that how the non-daemon thread completes its execution after the termination of the main thread but here is something different from that. In this example, we have created a function **thread_1()** and thread **T** as same as above example but here after the creation of thread **T** we use **setDaemon()** method to change the non-daemon nature of thread **T** to daemon nature, then we start the thread **T** and temporary stops the execution of the main thread. Here is the twist when the main thread is complete their execution and terminates then thread **T** also terminates because this is a daemon thread, where daemon thread terminates it's execution when the main thread terminates, work of it is to support the main thread if there is no main thread remaining why will daemon thread running there they also terminate still execution of instructions is remaining.

```
# Daemon Thread behaviour

from threading import *
import time

# creating a function
def thread_1():
    for i in range(5):
        print('this is thread T')
        time.sleep(3)

# creating a thread
T = Thread(target = thread_1)

# change T to daemon
T.setDaemon(True)   #setting a Daemon Thread

# starting of Thread T
T.start()
time.sleep(5)
print('this is Main Thread')
```