Threads and Processes

What Are Processes?

- A *process* (sometimes called a *heavyweight process*) is a program in execution.
- Each process has its own address space, memory, a data stack, and other auxiliary data to keep track of execution.
- The operating system manages the execution of all processes on the system, dividing the time fairly between all processes.
- Processes can also *fork* or *spawn* new processes to perform other tasks, but each new process has its own memory, data stack, etc., and cannot generally share information unless interprocess communication (IPC) is employed.

What Are Threads?

- *Threads* (sometimes called *lightweight processes*) are similar to processes except that they all execute within the same process, and thus all share the same context.
- They can be thought of as "miniprocesses" running in parallel within a main process or "main thread."
- A thread has a beginning, an execution sequence, and a conclusion. It can be preempted (interrupted) and temporarily put on hold (also known as *sleeping*) while other threads are running this is called *yielding.*
- If two or more threads access the same piece of data, inconsistent results may arise because of the ordering of data access. This is commonly known as a *race condition*

Python Threading Modules

- Python provides several modules to support MT programming, including the tHRead, tHReading, and Queue modules.
- The thread and threading modules allow the programmer to create and manage threads.
- The thread module provides basic thread and locking support, while threading provides higher-level,fully featured thread management.
- The Queue module allows the user to create a queue data structure that can be shared across multiple threads.
- However, the thread module has long been deprecated. Starting with Python 3, it has been designated as obsolete and is only accessible as __**thread** for backward compatibility.

❖ Exiting Threads

- When a thread completes execution of the function it was created for, it exits.
- Threads may also quit by calling an exit function such as tHRead. exit(), or any of the standard ways of exiting a Python process, i.e., sys.exit() or raising the SystemExit exception.

thread Module and its Lock Objects

Function/Method	*Description*
tHRead *Module Functions*	
start_new_thread(*function, args, kwargs=None)*	Spawns a new thread and execute *function* with the given *args* and optional *kwargs*
allocate_lock()	Allocates LockType lock object
exit()	Instructs a thread to exit
LockType *Lock Object Methods*	
acquire(*wait*=None)	Attempts to acquire lock object
locked()	Returns True if lock acquired, False otherwise
release()	Releases lock

tHReading Module

tHReading *Module Objects Description*

Thread	Object that represents a single thread of execution
Lock	Primitive lock object (same lock object as in the tHRead module)

RLock	Re-entrant lock object provides ability for a single thread to (re)acquire an already-held lock (recursive locking)
Condition	Condition variable object causes one thread to wait until a certain "condition" has been satisfied by another thread, such as changing of state or of some data value EventGeneral version of condition variables whereby any number of threads are waiting for some event to occur and all will awaken when the event happens
Semaphore	Provides a "waiting area"-like structure for threads waiting on a lock
BoundedSemaphore	Similar to a Semaphore but ensures it never exceeds its initial value Timer Similar to Thread except that it waits for an allotted period of time before running

threading Module Functions

Function	*Description*
activeCount()	Number of currently active Thread objects
current_Thread()	Returns the current THRead object
enumerate()	Returns list of all currently active Threads
settrace*(func)*	Sets a trace *function* for all threads
setprofile*(func)*	Sets a profile *function* for all threads

Thread Class

The THRead class of the threading is your primary executive object. It has a variety of functions not available to the thread module **Table**

Thread Object Methods

Method	*Description*
start()	Begin thread execution

run()	Method defining thread functionality (usually overridden by application writer in a subclass)
join(*timeout* = None)	Suspend until the started thread terminates; blocks unless *timeout* (in seconds) is given
getName()	Return name of thread
setName*(name)*	Set name of thread
isAlive()	Boolean flag indicating whether thread is still running
isDaemon()	Return daemon flag of thread
setDaemon*(daemonic)*	Set the daemon flag of thread as per the Boolean *daemonic* (must be called before thread start()ed)

Creating threads using Thread Module :

The syntax to create a new thread using this module is as follows:

_thread.**start_new_thread**(*function*, *args*[, *kwargs*])-Starts a new thread and return its identifier.

- The thread executes the function *function* with the argument list *args* (which must be a tuple).
- The optional *kwargs* argument specifies a dictionary of keyword arguments.
- When the function returns, the thread silently exits.

Example:

```
import time
import _thread

def thread_test(name, wait):
  i = 0
  while i <= 3:
    time.sleep(wait)
    print("Running %s\n" %name)
```

```
        i = i + 1

    print("%s has finished execution" %name)

if __name__ == "__main__":

    _thread.start_new_thread(thread_test, ("First Thread", 1))
    _thread.start_new_thread(thread_test, ("Second Thread", 2))
    _thread.start_new_thread(thread_test, ("Third Thread", 3))
```

OUTPUT:

Creating threads with Threading Module:

- In the threading module the most popular and the most used class is the Thread class, which is primarily used to create and run threads.
- Thread class provides all the major functionalities required to create and manage a thread.
- Thread objects are the objects of the Thread class where each object represents an activity to be performed in a separate thread of control.

There are two ways to create the Thread object and specify the activity to be performed:

- by passing a callable object to the constructor
- or, by overriding the run() method in a subclass.

Thread object which is created using constructor or run method can be started by using start() method.

start() method

This method is used to start the thread's activity. When we call this method, internally the run() method is invoked which executes the target function or the callable object.

Whenever a Thread object starts a new thread then internally it's run() method is invoked.It represents the thread's activity.

You may override this method in a subclass extending the Thread class of the threading module.

The standard run() method invokes the callable object passed to the object's constructor as the target argument with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

Thread class Constructor

Following is the basic syntax of the Thread class constructor:

```
Thread(group=None, target=None, name=None, args=(), kwargs={})
```

The constructor allows many arguments, some of which are required while some are not. Let's see what they are:

- group: Should be **None**. It is reserved for future extension.
- target: This is the callable object or task to be invoked by the run() method. As you can see in the code example at the top, we have specified the function names **thread1** and **thread2** as the value for this argument. It's default value is **None**.
- name: This is used to specify the thread name. By default, a unique name is generated following the format **Thread-N**, where **N** is a small decimal number.
- args: This is the argument **tuple** for the target invocation. We can provide values in it which can be used in the traget method. It's default value is empty, i.e. ()
- kwargs: This is keyword argument **dictionary** for the target invocation. This defaults to {}.

Example using Thread class object for creating a thread:

```
import threading

import time

def thread1(i):

    time.sleep(3)

    print('No. printed by Thread 1: %d' %i)

def thread2(i):

    print('No. printed by Thread 2: %d' %i)

 if __name__ == '__main__':

     t1 = threading.Thread(target=thread1, args=(10,))

    t2 = threading.Thread(target=thread2, args=(12,))

    # start the threads

    t1.start()

    t2.start()

    # join the main thread

    t1.join()

    t2.join()
```

OUTPUT:

Example using Thread subclass,overridingthe run() for creating a thread:

```
import threading

import time
```

```
class MyThread(threading.Thread):
  # overriding constructor
  def __init__(self, i):
    # calling parent class constructor
    threading.Thread.__init__(self)
    self.x = i

  # define your own run method
  def run(self):
    print("Value stored is: ", self.x)
    time.sleep(3)
    print("Exiting thread with value: ", self.x)
thread1 = MyThread(1)
thread1.start()
thread2 = MyThread(2)
thread2.start()
```

OUTPUT:

Python Thread.is_alive() Method

- **Thread.is_alive() method** is an inbuilt method of the *Thread* class of the *threading* module in Python.
- It uses a Thread object, and checks whether that thread is alive or not, ie, it is still running or not.
- This method returns True before the run() starts until just after the run() method is executed.

Module:

```
from threading import Thread
```

Syntax:

```
is_alive()
```

Parameter(s):

- None

Return value:

The return type of this method is `<class 'bool'>`, it returns True is the thread is alive else returns False.

Example:

```
#tt4.py
# Python program to explain the
# use of is_alive() method

import time
import threading

def thread_1(i):
    time.sleep(5)
    print('Value by Thread 1:', i)

def thread_2(i):
    print('Value by Thread 2:', i)

# Creating three sample threads
thread1 = threading.Thread(target=thread_1, args=(1,))
thread2 = threading.Thread(target=thread_2, args=(2,))

# Before calling the start(), both threads are not alive
print("Is thread1 alive:", thread1.is_alive())
print("Is thread2 alive:", thread2.is_alive())
print()
```

```
thread1.start()
thread2.start()
# Since thread11 is on sleep for 5 seconds, it is alive
# while thread 2 is executed instantly

print("Is thread1 alive:", thread1.is_alive())
print("Is thread2 alive:", thread2.is_alive())
```

Output

```
Is thread1 alive: False
Is thread2 alive: False

Value by Thread 2: 2
Is thread1 alive: True
Is thread2 alive: False
Value by Thread 1: 1
```

Python Thread.getName() Method

Thread.getName() method is an inbuilt method of the *Thread* class of the *threading* module in Python.

This method is used to get the name of the thread.

Module:

```
from threading import Thread
```

Syntax:

```
getName()
```

Parameter(s):

- None

Return value:

The return type of this method is `<class 'str'>`, it returns thread name of the calling object.

Example:

```
#tt5.y
# Python program to explain the
# use of getName() method

import time
import threading

def thread_1(i):
    time.sleep(5)
    print('Value by '+ str(threading.current_thread().getName())+" is: ", i)

def thread_2(i):
    print('Value by '+ str(threading.current_thread().getName())+" is: ", i)

def thread_3(i):
    time.sleep(4)
    print('Value by '+ str(threading.current_thread().getName())+" is: ", i)

# Creating three sample threads
thread1 = threading.Thread(target=thread_1, args=(10,))
thread2 = threading.Thread(target=thread_2, args=(20,))
thread3 = threading.Thread(target=thread_2, args=(30,))

# Running the threads
thread1.start()
thread2.start()
thread3.start()
```

Output

```
Value by Thread-2 is:  20
Value by Thread-3 is:  30
Value by Thread-1 is:  10
```

Python Thread.setName() Method

Thread.setName() method is an inbuilt method of the *Thread* class of the *threading* module in Python. It uses a *Thread* object and sets the name of the thread.

Module:

```
from threading import Thread
```

Syntax:

```
setName()
```

Parameter(s):

- None

Return value:

The return type of this method is `<class 'NoneType'>`, it sets the name of the Thread object which calls this method.

Example:

```
#tt6.py
# Python program to explain the
# use of setName() method

import time
import threading

def thread_1(i):
    time.sleep(5)
    #threading.current_thread.setName("frgrfvrv")
    print('Value by '+ str(threading.current_thread().getName())+" is: ", i)

def thread_2(i):
    print('Value by '+ str(threading.current_thread().getName())+" is: ", i)

def thread_3(i):
    time.sleep(4)
    print('Value by '+ str(threading.current_thread().getName())+" is: ", i)

# Creating three sample threads
thread1 = threading.Thread(target=thread_1, args=(10,))
thread1.setName("Thread_number_1")
thread2 = threading.Thread(target=thread_2, args=(20,))
```

```
thread2.setName("Thread_number_2")
thread3 = threading.Thread(target=thread_2, args=(30,))
thread3.setName("Thread_number_3")

# Running the threads
thread1.start()
thread2.start()
thread3.start()
```

Output

```
Value by Thread_number_2 is:  20
Value by Thread_number_3 is:  30
Value by Thread_number_1 is:  10
```

Python threading.active_count() Method

active_count() is an inbuilt method of the *threading* module in Python.

It is used to return the number of Thread objects that are active at any instant.

Module:

```
import threading
```

Syntax:

```
active_count()
```

Parameter(s):

- None

Return value:

The return type of this method is <class 'int'>, it returns the number of active Thread class objects at any instant.

Example:

```
#tt9.y
# Python program to explain the use of
# active_count() method in Threading Module

import time
import threading

def thread_1(i):
    time.sleep(2)
    print("Number of active threads:", threading.active_count())
    print('Value by Thread 1:', i)

def thread_2(i):
    time.sleep(5)
    print("Number of active threads:", threading.active_count())
    print('Value by Thread 2:', i)

def thread_3(i):
    print("Number of active threads:", threading.active_count())
    print("Value by Thread 3:", i)

# Creating sample threads
thread1 = threading.Thread(target=thread_1, args=(1,))
thread2 = threading.Thread(target=thread_2, args=(2,))
thread3 = threading.Thread(target=thread_3, args=(3,))

print("Number of active threads in the starting:",
threading.active_count())
print("The active threads in the starting is 1 which is the main thread
that executes till the program runs")

# Starting the threads
thread1.start()
thread2.start()
thread3.start()
```

Output

```
Number of active threads in the starting: 2
The active threads in the starting is 1 which is the main thread that
executes till the program runs
Number of active threads:
```

```
>>> 5
Value by Thread 3: 3
Number of active threads: 4
Value by Thread 1: 1
Number of active threads: 3
Value by Thread 2: 2
```

Python threading.current_thread() Method

current_thread() is an inbuilt method of the *threading* module in Python. It is used to return the current Thread object, which corresponds to the caller's thread of control.

Module:

```
import threading
```

Syntax:

```
current_thread()
```

Parameter(s):

- None

Return value:

The return type of this method is a Thread class object, it returns the current Thread object active at the moment.

Example:

```
#t10.py
# Python program to explain the use of
# current_thread() method in Threading Module

import time
import threading

def thread_1(i):
    time.sleep(2)
    print("Active current thread right now:", (threading.current_thread()))
    print('Value by Thread 1:', i)

def thread_2(i):
    time.sleep(5)
    print("Active current thread right now:", (threading.current_thread()))
    print('Value by Thread 2:', i)

def thread_3(i):
    print("Active current thread right now:", (threading.current_thread()))
    print("Value by Thread 3:", i)

# Creating sample threads
thread1 = threading.Thread(target=thread_1, args=(1,))
thread2 = threading.Thread(target=thread_2, args=(2,))
thread3 = threading.Thread(target=thread_3, args=(3,))

print("Active current thread right now:", (threading.current_thread()))
#3 Initially it is the main thread that is active

# Starting the threads
thread1.start()
thread2.start()
thread3.start()
```

Output

```
Active current thread right now: <_MainThread(MainThread, started
14392)>
Active current thread right now:
>>> <Thread(Thread-3, started 11380)>
Value by Thread 3: 3
Active current thread right now: <Thread(Thread-1, started 14768)>
Value by Thread 1: 1
Active current thread right now: <Thread(Thread-2, started 14900)>
Value by Thread 2: 2
```

Python threading.enumerate() Method

enumerate() is an inbuilt method of the *threading* module in Python. It is used to return the list of all the Thread class objects which are currently alive.

It also includes daemonic threads, the main thread, and dummy thread objects created by current_thread(). It does not count the threads that have terminated or which have not started yet.

Module:

```
import threading
```

Syntax:

```
enumerate()
```

Parameter(s):

- None

Return value:

The return type of this method is `<class 'list'>`, it returns a list of the currently alive Thread class objects.

Example:

```
#tt11.py
# Python program to explain the use of
# enumerate()  method in the Threading Module

import time
import threading

def thread_1(i):
    time.sleep(5)
    print("Threads alive when thread_1 executes:")
    print(*threading.enumerate(), sep = "\n")

    print()
```

```
def thread_2(i):
    print("Threads alive when thread_2 executes")
    print(*threading.enumerate(), sep = "\n")
    print()

def thread_3(i):
    time.sleep(4)


def thread_4(i):
    time.sleep(1)
    print("Threads alive when thread_4 executes")
    print(*threading.enumerate(), sep = "\n")
    print()

# Creating sample threads
thread1 = threading.Thread(target=thread_1, args=(10,))
thread2 = threading.Thread(target=thread_2, args=(20,))
thread3 = threading.Thread(target=thread_3, args=(30,))
thread4 = threading.Thread(target=thread_4, args=(50,))

print("Threads alive in the starting:", threading.enumerate())
print()

# Starting the threads
thread1.start()
thread2.start()
thread3.start()
thread4.start()
```

OUTPUT:

```
Threads alive in the starting: [<_MainThread(MainThread, started
13332)>, <Thread(SockThread, started daemon 10824)>]

Threads alive when thread_2 executes
>>> <_MainThread(MainThread, started 13332)>
<Thread(SockThread, started daemon 10824)>
<Thread(Thread-1, started 18120)>
<Thread(Thread-2, started 5384)>
<Thread(Thread-3, started 14776)>
<Thread(Thread-4, started 1804)>
```

```
Threads alive when thread_4 executes
<_MainThread(MainThread, started 13332)>
<Thread(SockThread, started daemon 10824)>
<Thread(Thread-1, started 18120)>
<Thread(Thread-3, started 14776)>
<Thread(Thread-4, started 1804)>

Threads alive when thread_1 executes:
<_MainThread(MainThread, started 13332)>
<Thread(SockThread, started daemon 10824)>
<Thread(Thread-1, started 18120)>
```

Synchronizing threads

- To deal with race conditions, deadlocks, and other thread-based issues, the threading module provides the **Lock** object.
- The idea is that when a thread wants access to a specific resource, it acquires a lock for that resource.
- Once a thread locks a particular resource, no other thread can access it until the lock is released.
- As a result, the changes to the resource will be atomic, and race conditions will be averted.
- A lock is a low-level synchronization primitive implemented by the __**thread** module.

At any given time, a lock can be in one of 2 states: **locked** or **unlocked.** It supports two methods:

1. **acquire()**When the lock-state is unlocked, calling the acquire() method will change the state to locked and return. However, If the state is locked, the call to acquire() is blocked until the release() method is called by some other thread.
2. **release()**The release() method is used to set the state to unlocked, i.e., to release a lock. It can be called by any thread, not necessarily the one that acquired the lock.

```
import threading
lock = threading.Lock()

def first_function():
    for i in range(5):
        lock.acquire()
        print ('lock acquired')
        print ('Executing the first funcion')
        lock.release()

def second_function():
    for i in range(5):
        lock.acquire()
        print ('lock acquired')
        print ('Executing the second funcion')
        lock.release()

if __name__=="__main__":
    thread_one = threading.Thread(target=first_function)
    thread_two = threading.Thread(target=second_function)

    thread_one.start()
    thread_two.start()

    thread_one.join()
    thread_two.join()
```

Apart from locks, python also supports some other mechanisms to handle thread synchronization as listed below:

1. RLocks
2. Semaphores
3. Conditions
4. Events, and
5. Barriers

Global Interpreter Lock (GIL)

- In python GIL is a process lock or a mutex used while dealing with the processes.
- It makes sure that one thread can access a particular resource at a time and it also prevents the use of objects and bytecodes at once.
- This benefits the single-threaded programs in a performance increase. GIL in python is very simple and easy to implement.
- A lock can be used to make sure that only one thread has access to a particular resource at a given time.

- One of the features of Python is that it uses a global lock on each interpreter process, which means that every process treats the python interpreter itself as a resource.

For example, suppose you have written a python program which uses two threads to perform both CPU and ‘I/O’ operations. When you execute this program, this is what happens:

1. The python interpreter creates a new process and spawns the threads
2. When thread-1 starts running, it will first acquire the GIL and lock it.
3. If thread-2 wants to execute now, it will have to wait for the GIL to be released even if another processor is free.
4. Now, suppose thread-1 is waiting for an I/O operation. At this time, it will release the GIL, and thread-2 will acquire it.
5. After completing the I/O ops, if thread-1 wants to execute now, it will again have to wait for the GIL to be released by thread-2.

Due to this, only one thread can access the interpreter at any time, meaning that there will be only one thread executing python code at a given point of time.

Why was GIL needed?

The CPython garbage collector uses an efficient memory management technique known as reference counting. Here’s how it works: Every object in python has a reference count, which is increased when it is assigned to a new variable name or added to a container (like tuples, lists, etc.). Likewise, the reference count is decreased when the reference goes out of scope or when the del statement is called. When the reference count of an object reaches 0, it is garbage collected, and the allotted memory is freed.

But the problem is that the reference count variable is prone to race conditions like any other global variable. To solve this problem, the developers of python decided to use the global interpreter lock. The other option was to add a lock to each object which would have resulted in deadlocks and increased overhead from acquire() and release() calls.

Therefore, GIL is a significant restriction for multithreaded python programs running heavy CPU-bound operations (effectively making them single-threaded). If you want to make use of multiple CPU cores in your application, use the **multiprocessing** module instead.

Summary

Python supports 2 modules for multithreading:

1. __**thread** module: It provides a low-level implementation for threading and is obsolete.
2. **threading module**: It provides a high-level implementation for multithreading and is the current standard.

To create a thread using the threading module, you must do the following:

3. Create a class which extends the **Thread** class.
4. Override its constructor (__init__).
5. Override its **run()** method.
6. Create an object of this class.

- A thread can be executed by calling the **start()** method.
- The **join()** method can be used to block other threads until this thread (the one on which join was called) finishes execution.
- A race condition occurs when multiple threads access or modify a shared resource at the same time.
- It can be avoided by Synchronizing threads.

Python supports 6 ways to synchronize threads:

1. Locks
2. RLocks
3. Semaphores
4. Conditions
5. Events, and
6. Barriers

Locks allow only a particular thread which has acquired the lock to enter the critical section.

A Lock has 2 primary methods:

1. **acquire()**: It sets the lock state to **locked.** If called on a locked object, it blocks until the resource is free.

2. **release()**: It sets the lock state to **unlocked** and returns. If called on an unlocked object, it returns false.

- The global interpreter lock is a mechanism through which only 1 CPython interpreter process can execute at a time.
- It was used to facilitate the reference counting functionality of CPythons's garbage collector.
- To make Python apps with heavy CPU-bound operations, you should use the multiprocessing module.

Multithreaded Priority Queue

- The Queue module is primarily used to manage to process large mounts of data on multiple threads.
- It supports the creation of a new queue object that can take a distinct number of items.
- The get() and put() methods are used to add or remove items from a queue respectively.

Below is the list of operations that are used to manage Queue:

- **get():** It is used to add an item to a queue.
- **put():** It is used to remove an item from a queue.
- **qsize():** It is used to find the number of items in a queue.
- **empty():** It returns a boolean value depending upon whether the queue is empty or not.
- **full():** It returns a boolean value depending upon whether the queue is full or not.

A Priority Queue is an extension of the queue with the following properties:

- An element with high priority is dequeued before an element with low priority.
- If two elements have the same priority, they are served according to their order in the queue.

Below is a code example explaining the process of creating multi-threaded priority queue:

Example

```
#t8.py
```

```
import queue
import threading
import time

thread_exit_Flag = 0

class sample_Thread (threading.Thread):
   def __init__(self, threadID, name, q):
      threading.Thread.__init__(self)
      self.threadID = threadID
      self.name = name
      self.q = q
   def run(self):
      print ("initializing " + self.name)
      process_data(self.name, self.q)
      print ("Exiting " + self.name)

# helper function to process data
def process_data(threadName, q):
   while not thread_exit_Flag:
      queueLock.acquire()
      if not workQueue.empty():
         data = q.get()
         queueLock.release()
         print ("% s processing % s" % (threadName, data))
      else:
         queueLock.release()
         time.sleep(1)

thread_list = ["Thread-1", "Thread-2", "Thread-3"]
name_list = ["A", "B", "C", "D", "E"]
queueLock = threading.Lock()
workQueue = queue.Queue(10)
threads = []
threadID = 1

# Create new threads
for thread_name in thread_list:
  thread = sample_Thread(threadID, thread_name, workQueue)
  thread.start()
  threads.append(thread)
  threadID += 1

# Fill the queue
queueLock.acquire()
for items in name_list:
  workQueue.put(items)

queueLock.release()

# Wait for the queue to empty
while not workQueue.empty():
  pass

# Notify threads it's time to exit
thread_exit_Flag = 1

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

When the above code is executed, it produces the following result –

OUTPUT:

```
initializing Thread-1initializing Thread-3
initializing Thread-2
Thread-1 processing BThread-3 processing AThread-2 processing C
Thread-3 processing D
Thread-1 processing Exiting Thread-3
Exiting Thread-1
Exiting Thread-2
Exit Main Thread
```

Related Modules of threads:

The table below lists some of the modules you may use when programming multithreaded applications.

Threading-Related Standard Library Modules

Module	***Description***
tHRead	Basic, lower-level thread module
threading	Higher-level threading and synchronization objects
Queue	Synchronized FIFO queue for multiple threads
mutex	Mutual exclusion objects
SocketServer	TCP and UDP managers with some threading control

All functions in single program

```
import time
import threading

def thread_1(i):
   time.sleep(5)
   #print('Value by Thread 1:', i)
   print('Value by '+ str(threading.current_thread().getName())+" is: ", i)
   print("Threads alive in first:", threading.enumerate())
   print("Number of active threads including first thread:",
threading.active_count())
   print("Active current thread right now:", (threading.current_thread()))

def thread_2(i):
   #print('Value by Thread 2:', i)
   print('Value by '+ str(threading.current_thread().getName())+" is: ", i)
   print("Threads alive in second:", threading.enumerate())
   print("Number of active threads including second thread:",
threading.active_count())
   print("Active current thread right now:", (threading.current_thread()))

# Creating three sample threads
thread1 = threading.Thread(target=thread_1, args=(1,))
thread1.setName("First Thread")
```

```
thread2 = threading.Thread(target=thread_2, args=(2,))
thread2.setName("Second Thread")
print("Threads alive in the starting:", threading.enumerate())
print("Number of active threads in the starting:", threading.active_count())
print("Active current thread right now:", (threading.current_thread()))

# Before calling the start(), both threads are not alive
print("Is thread1 alive:", thread1.is_alive())
print("Is thread2 alive:", thread2.is_alive())

thread1.start()
thread2.start()
# Since thread11 is on sleep for 5 seconds, it is alive
# while thread 2 is executed instantly

print("Is First thread alive:", thread1.is_alive())
print("Is Second thread alive:", thread2.is_alive())
print("Number of active threads before join:", threading.active_count())
thread1.join()
thread2.join()
print("Threads alive in the ending:", threading.enumerate())
print("Active current thread right now:", (threading.current_thread()))
print("Number of active threads after join ending:",
threading.active_count())
```

OUTPUT:

```
Threads alive in the starting: [<_MainThread(MainThread, started 924)>,
<Thread(SockThread, started daemon 1352)>]
Number of active threads in the starting: 2
Active current thread right now: <_MainThread(MainThread, started
924)>
Is thread1 alive: False
Is thread2 alive: False
Value by Second Thread is: Is First thread alive:  2True

Threads alive in second:Is Second thread alive:
[<_MainThread(MainThread, started 924)>, <Thread(SockThread,
started daemon 1352)>, <Thread(First Thread, started 6464)>,
<Thread(Second Thread, started 12796)>]True

Number of active threads including second thread:Number of active
threads before join:  44
```

```
Active current thread right now: <Thread(Second Thread, started 12796)>
Value by First Thread is:  1
Threads alive in first: [<_MainThread(MainThread, started 924)>,
<Thread(SockThread, started daemon 1352)>, <Thread(First Thread,
started 6464)>]
Number of active threads including first thread: 3
Active current thread right now: <Thread(First Thread, started 6464)>
Threads alive in the ending: [<_MainThread(MainThread, started 924)>,
<Thread(SockThread, started daemon 1352)>]
Active current thread right now: <_MainThread(MainThread, started
924)>
Number of active threads after join ending: 2
```