Multithreading in Python

Almost all of us have a computer or a laptop. The tasks we do using these devices are controlled by one of the parts called the CPU (Central Processing Unit). Multithreading is the concept related to how the tasks are executed.

In this article, you will learn about multithreading, implementing this concept in Python using the threading module. We will also see multiple functions in this module. So, let us start with the introduction to multithreading.

Introduction to Multithreading

Before learning multithreading, let us see what does the word ‘thread’ means.

A process in a computer/laptop is an instance of the program being executed. A process contains the program to be executed, the corresponding variables, and its state during the execution.

A thread is a sequence of instructions that an operating system can execute independently. It can be thought of as the smallest unit of the process.

Previously, a device could handle only one task at a time. But now most of the devices implement multithreading where the CPUs allow execution of multiple threads/ tasks at a time independently. These threads contain their own register set and local variables. Since these threads under a process are in the same data space, they can share global variables and the code.

Multithreading in Python

We can do multithreading in Python, that is, executing multiple parts of the program

at a time using the threading module. We can import this module by writing the below statement.

```
1.  import threading
```

This module has a higher class called the Thread(), which handles the execution of the program as a whole. We can create multiple instances of this class to handle different parts of the code concurrently. For example,

Example of multithreading in Python:

```
1.  import threading
2.
3.  def even():#creating second function
4.      for i in range(0,20,2):
5.          print(i)
6.  def odd():
7.      for i in range(1,20,2):
8.          print(i)
9.
10. # creating a thread for each function
11. trd1 = threading.Thread(target=even)
12. trd2 = threading.Thread(target=odd)
13.
14. trd1.start() # starting the thread 1
15.
16. trd2.start() # starting the thread 2
17.
18. print('End')
```

Output:

```
0
2
4
6
8
10
12
141
End
3
5
7
9
11
13
15
17
16
18
19
```

In this program,

1. We imported the module
2. Created two functions. One to print even numbers and the other to print odd numbers.
3. We created the instances of the Thread class for each of the functions.

4. Then we executed the functions using the start() function in the Thread class.
5. Then we wrote the print statement that displays 'End'

From the output, we can see that results of the functions and the 'End' statement did not get printed one after another. The result is not the same as the usual program where the execution takes place sequentially. Also, in the line with the output 141, we can see that the output 14 from the even() function and the output 1 from the odd() function got printed on the same line. These show that these two threads are executing simultaneously.

One thing to be remembered is that all these threads of a program that belong to the same process are under the main thread that handles the execution of the same program.

We can set the 'End' print statement to be executed after both the threads are done

with the execution. We can do this by using the join() function as shown below.

Example of multithreading in Python:

```
1.  import threading
2.
3.  def even():#creating a function
4.      for i in range(0,20,2):
5.          print(i)
6.  def odd():
7.      for i in range(1,20,2):
8.          print(i)
9.
10.  # creating thread
11.  trd1 = threading.Thread(target=even)
12.  trd2 = threading.Thread(target=odd)
13.
14.  trd1.start() # starting the thread 1
15.
16.  trd2.start() # starting the thread 2
17.
18.  # waiting until thread 1 is done with the
    execution
19.  trd1.join()
20.  # waiting until thread 2 is done with the
    execution
21.  trd2.join()
22.
23.  # both threads completely executed
24.  print("End")
```

Output:

```
0
2
4
```

```
6
81
3
5
7
9
11
13
15
17
1910
12
14
16
18
End
```

Functions in Python Multithreading

Python provides multiple functions that can be used while doing multithreading. Let us see each of them.

1. active_count():

This function returns the number of currently active Thread objects. This value

is equal to the length of the list that the function enumerate() returns. For example,

Example of active_count():

```
1.  threading.active_count()
```

Output:

5

2. current_thread():

This function returns the current Thread object. The thread returned will depend on the thread of control at the caller's end. If this thread isn't through 'threading', it returns a dummy thread object that has limited functionality. For example,

Example of current_thread():

```
1.  threading.current_thread()
```

Output:

<_MainThread(MainThread, started 15780)>

3. enumerate():

This returns a list of Thread objects that are currently active. This includes the main thread, daemonic threads, and dummy thread objects created by current_thread(). For example,

Example of enumerate():

```
1.  threading.enumerate()
```

Output:

[<_MainThread(MainThread, started 15780)>,
<Thread(Thread-4, started daemon 9060)>,
<Heartbeat(Thread-5, started daemon 10000)>,
<HistorySavingThread(IPythonHistorySavingThread, started 17164)>,
<ParentPollerWindows(Thread-3, started daemon 11660)>]

4. get_ident():

This function returns the identifier of the current thread. We can use this identifier as

an index of a dictionary to get thread-specific data.

Example of get_ident():

```
1.  threading.get_ident()
```

Output:

15780

5. main_thread():

This function returns the main Thread object. Normally, it is that thread that started the interpreter.

Example of main_thread():

```
1.  threading.main_thread()
```

Output:

```
<_MainThread(MainThread, started 15780)>
```

6. settrace(func):

This function traces a function for all threads created using the 'threading' module. The function is passed as the

argument to this settrace(), before it calls its run() method. Its syntax is

```
threading.settrace(func)
```

7. setprofile(func):

This method sets a profile function for all threads we started from 'threading'. It passes func to sys.setprofile() for each thread before it calls its run() method. Its syntax is

```
threading.setprofile(func)
```

8. stack_size([size]):

This function returns the thread stack size on creating new threads. The size is optional and must be equal to 0 or a positive integer, by default it is 0.

On passing an invalid size we get a ValueError. The minimum value of stack size in most of the systems is 32 KiB. If it doesn't support changing the thread stack size, we get a RuntimeError. Its syntax is

```
threading.stack_size()
```

9. TIMEOUT_MAX:

This is a constant value in this module that holds the maximum allowed value for the timeout parameter for blocking functions like Lock.acquire(), Condition.wait(), RLock.acquire(), and others. We can get this value as shown below.

Example of TIMEOUT_MAX:

```
threading.TIMEOUT_MAX
```

Output:

```
4294967.0
```

Thread Local Data

The data specific to a thread is known as Thread Local Data. To create local data we need to first create an instance of local subclass in Thread. And then store the data as an attribute. For example,

Example of creating Thread local data

```
data=threading.local()
data.name='ABC'
```

These data are specific to each thread in a program and are handled by the class threading.class.

Thread Objects in Python

In the example of multithreading using Python, we saw how we create the instances of the Thread class. We also saw some functions in this class and their use. We will discuss the related functions of this class in this section.

First let us see the syntax of the Thread class.

```
class threading.Thread(group=None, target=None, name=None, args=(), kwargs={}, *, daemon=None)
```

In this syntax,
1. The value of the group must be None. This is reserved for the extension when we implement a ThreadGroup class.
2. The target argument takes a callable

object that run() will invoke. By default, it is None, which means it invokes nothing.
3. The name is the name of the thread. The default for this is “Thread-N”.
4. The args is a tuple containing the arguments needed to be passed while invoking the target. The default value is an empty tuple.
5. The kwargs is a dictionary containing the keyword arguments. This also helps in invoking the target and its default is an empty dictionary.
6. The daemon decides whether the thread is daemonic or not. The default value is None, and in this case, it inherits the daemonic property from the current thread.

It is recommended to give the keyword arguments to not get errors on wrong ordering. Also, it is needed to be made sure to invoke the base class constructor(Thread.__init__()) first if the subclass overrides the constructor.

And the methods inside the Thread class are:

1. start()

This function starts the thread activity. We get a RuntimeError if we call it more than once. So,it should be called only once for a thread instance. This is shown in the below example,

Example of start() giving error on calling again:

```
1.  threading.Thread.start(threading.current_
    )
```

Output:

```
RuntimeError Traceback (most recent call
last)
<ipython-input-10-787da1f98892> in <module>
—-> 1
threading.Thread.start(threading.current_th
read())~\anaconda3\lib\threading.py in
start(self)
846
847 if self._started.is_set():
-> 848 raise RuntimeError(“threads can only
be started once”)
```

```
849 with _active_limbo_lock:
850 _limbo[self] = selfRuntimeError:
threads can only be started once
```

In this example, we are calling the currently active thread again, so we are getting an error.

2. run()

This function invokes the callable object we passed to the constructor as the target argument if it exists. While calling the target, the args and kwargs are given to the object. We can override this function in a subclass.

3. join(timeout=None)

This function is used when we want the program to wait until the thread execution is completed. It blocks the calling of thread until the one on which we call join() terminates normally or with an exception, or until timeout is reached.

To pass a timeout in seconds or fractions, we can give a floating value timeout. This function returns None. So, to know if a timeout occurred or not we need to use the is_alive() function. If timeout is None, then join() works till the thread's execution is done or till we get any error.

We are allowed to join() a thread multiple times. But if we try to join the current thread, we get a RuntimeError because this causes a deadlock. This is shown in the below example.

Example of join() giving error on joining current thread:

```
1.  threading.Thread.join(threading.current_t
    )
```

Output:

```
RuntimeError Traceback (most recent call
last)
<ipython-input-11-f97dde42ddb8> in <module>
--> 1
threading.Thread.join(threading.current_thr
ead())~\anaconda3\lib\threading.py in
```

```
join(self, timeout)
1006 raise RuntimeError("cannot join thread
before it is started")
1007 if self is current_thread():
-> 1008 raise RuntimeError("cannot join
current thread")
1009
1010 if timeout is None:RuntimeError:
cannot join current thread
```

4. name:

This is a string that is used for identification. We can give the same name to multiple threads. The constructor sets the initial name. We can set the name using the below syntax.

```
1.  threading.Thread.name='name'
```

5. getName() and setName()

These functions are the old getter and setter APIs for getting and setting the name.

6. ident

On starting a thread, either an identifier or None is returned. The identifier is a non-

zero integer, similar to the get_ident() function.

7. is_alive()

This function is used to check if the thread is alive or not. It returns true for a thread if it is in the phase between the starting of the run() function and the stage after it terminates. We can check it as shown below.

Example of is_alive() giving error on joining current thread:

```
1.  threading.Thread.is_alive(threading.curre
    )
```

Output:

```
True
```

This returns True as we checked for the current thread which is active.

8. daemon

This is a Boolean value that tells us if the thread is a daemon or not. The main thread is not a daemon. Hence, all threads in the main thread have a default value of False for the daemon. So, we must set it before we call the start() function.

9. isDaemon() and setDaemon()

These are the getter and setter APIs for getting and setting the daemon attribute.

Python Locks

Before looking into the locks, it is important to know the problem of race conditions. This occurs in the situation when multiple threads try to modify the same shared data. Since the threads are running simultaneously, there is a lot of chance of ambiguous output or change. For more understanding look at the below example.

Example of race condition:

```
1. import threading
```

```
2.
3.  a=0
4.
5.  def inc():
6.      global a
7.      a += 1
8.
9.  def call_inc():
10.     for _ in range(200000):
11.         inc()
12.
13. def task():
14.     global a
15.
16.     a = 0
17.
18.     # creating the threads with the
    target as call_inc() function
19.     trd1 =
    threading.Thread(target=call_inc)
20.     trd2 =
    threading.Thread(target=call_inc)
21.
22.     # starting the threads
23.     trd1.start()
24.     trd2.start()
25.
26.     # waiting til thread 1 is done with
    the execution
27.     trd1.join()
28.     trd2.join()
29.
30.
31. for i in range(5):
32.     task()
33.     print(f"Iteration {i} and the value
    of a = {a}")
```

Output:

```
Iteration 0 and the value of a = 400000
Iteration 1 and the value of a = 294987
Iteration 2 and the value of a = 400000
```

```
Iteration 3 and the value of a = 302137
Iteration 4 and the value of a = 400000
```

Since the call_inc() is called twice using the two threads, the expected output is 400000 in all iterations, but we got other values. This happens due to the race condition.

We use the locks to solve this problem. The locks provide two methods and these are:

1. acquire(blocking=True, timeout=-1):

This function acquires a lock, which can be either blocking or non-blocking.
When the parameter blocking is True (the default), thread execution is blocked until the lock is unlocked. Then the lock is set to locked and returns True. But when it is False, thread execution is not blocked. If the lock is unlocked, then it is locked, and True is returned else returned False.

2. release():

To function releases the lock. When the lock is locked, it is reset to unlocked and

returned. If any other threads are blocked, waiting for the lock to become unlocked, it allows exactly one of them to proceed. If the lock is already unlocked, we get a ThreadError.

Now let us see the results of using the locks.

Example of lock:

```
import threading

a=0

def inc():
    global a
    a += 1

def call_inc(lock):
    for _ in range(200000):
        lock.acquire()
        inc()
        lock.release()

def task():
    global a

    a = 0

    #Creating a lock object
    lock = threading.Lock()

    # creating the threads with the targe
call_inc() function and passing the lock
object as argument
    trd1 =
threading.Thread(target=call_inc,args=(loc
    trd2 =
threading.Thread(target=call_inc,args=(loc
```

```
26.
27.       # starting the threads
28.       trd1.start()
29.       trd2.start()
30.
31.       # waiting til thread 1 is done with tl
execution
32.       trd1.join()
33.       trd2.join()
34.
35.
36.    for i in range(5):
37.       task()
38.       print(f"Iteration {i} and the value o
= {a}")
```

Output:

```
Iteration 0 and the value of a = 400000
Iteration 1 and the value of a = 400000
Iteration 2 and the value of a = 400000
Iteration 3 and the value of a = 400000
Iteration 4 and the value of a = 400000
```

Here, we created a lock object and passed it as an argument to the call_inc() function. And in the call_inc() function, we are applying lock using lock.acquire(), not letting other threads access this section. And then releasing the lock only when the increment is done.

Because, the threads do not race here, we get the 400000 in all iterations.

Python RLocks and Semaphore Locks

The RLock stands for Re-entrant lock. The normal lock does not recognize which thread holds the lock of the shared resources. Due to this, if the resource is blocked by the thread then the same thread is blocked if it tries to access again. In these situations, we use RLocks.

Multiple threads can access the shared resources any number of times. Similar to the Locks, these objects also have two methods:

1. acquire (blocking=True, timeout=-1)
2. release()

These have the same functionality as discussed in the Locks objects. For

example,

Example of lock:

```
import threading

a=0

#Creating a RLock object
lock = threading.RLock()


lock.acquire()
a = a + 1

# accessing the shared resource
lock.acquire()
a=a+3
lock.release()
lock.release()

# displaying the value of the shared resource
print(a)
```

Output:

```
4
```

Similar to the RLocks, there are Semaphore locks. A semaphore can be thought of as an extension to lock. It allows changing the resource first by checking if the resource is in use or not. If it is in use, then it lets the thread wait and if not, then it does the

required modification in the resource. Even it has two functions acquire() and release().

Advantages and Disadvantages of Multithreading in Python

Learning a lot about multithreading, let us see some pros and cons of using this concept.

The advantages include:

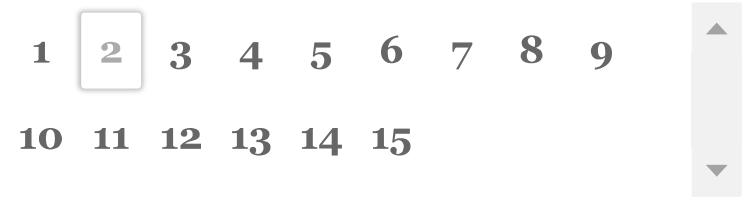
1. Since the threads are independent of each other, the user is not blocked.
2. Since the threads do parallel execution, the resources of the device can be used efficiently.
3. Enhance the performance of the multi-processor machines.
4. Many of the GUIs and CPUs used in the current days include multithreading.

The disadvantages are:

1. The complexity increases with the increase in the number of threads

2. It is necessary to maintain synchronization while sharing the resources among multiple threads
3. It is difficult to debug the programs with threads as the result is sometimes unpredictable.
4. The process of constructing and synchronizing the threads is CPU/memory intensive.

Quiz on Multithreading in Python



2. Question

- **Which of the following is true about the thread of a program?**

○ Every program has a main thread

○ All the threads have the same id

○ There can be more than one thread set to a program

○ All the above

Skip question Check

Conclusion

In this article, we saw the concepts of multithreading, doing the same in Python. Then, we saw the functions, objects in the threading module. After this, we saw the

concepts of synchronization using locks. Finally, we discussed some pros and cons of multithreading.

Hope the concepts covered are understood. Happy learning!

Tags: Advantages of Multithreading in Python Disadvantage of Multithreading in Python Functions in Python Multithreading Multithreading in Python Python Locks Thread Local Data Thread Objects in Python

LEAVE A REPLY

Comment *

Name * **Email** *

☐ **Save my name, email, and website in this browser for the next time I comment.**

Post Comment