Programming in Python

GAVRILUT DRAGOS

COURSE 11

Threading and Synchronization

Python threading support is achieved through two modules:

- **thread** ➔ old methods, low-level methods
- **threading** ➔ new methods, based on a class model

Details about these modules can be found on:

- Python 2: https://docs.python.org/2/library/thread.html
- Python 3: https://docs.python.org/3/library/_thread.html
- Python 2: https://docs.python.org/2/library/threading.html
- Python 3: https://docs.python.org/3/library/threading.html

Besides these a series of synchronization object that include locks, semaphores, events are also available.

As **thread** module was renamed in python 3 to **_thread** it is best to use threading if you want a code that will run in the same way in Python 2 and Python 3.

Threading and Synchronization

To start a new thread in python 2.x use **start_new_thread** method from class thread. The method receives a function that will be executed on the new thread and functions parameters.

Python 2.x

```
import thread,time

def MyPrint(sleepPeriod,name,count):
        for i in range(0,count):
                print (name+"=>"+str(i))
                time.sleep(sleepPeriod)

#main thread
thread.start_new_thread (MyPrint, (1,"Thread #1", 3))
thread.start_new_thread (MyPrint, (2,"Thread #2", 2))
thread.start_new_thread (MyPrint, (3,"Thread #3", 1))
time.sleep(10)
```

Output

```
Thread #1=>0Thread #2=>0

 Thread #3=>0
Thread #1=>1
Thread #2=>1
Thread #1=>2
```

Threading and Synchronization

To start a new thread in python 2.x use **start_new_thread** method from class thread. The method receives a function that will be executed on the new thread and functions parameters.

Python 3.x

```
import _thread,time

def MyPrint(sleepPeriod,name,count):
        for i in range(0,count):
                print (name+"=>"+str(i))
                time.sleep(sleepPeriod)

#main thread
_thread.start_new_thread (MyPrint, (1,"Thread #1", 3))
_thread.start_new_thread (MyPrint, (2,"Thread #2", 2))
_thread.start_new_thread (MyPrint, (3,"Thread #3", 1))
time.sleep(10)
```

Output

```
Thread #1=>0
Thread #2=>0
Thread #3=>0
Thread #1=>1
Thread #2=>1
Thread #1=>2
```

Threading and Synchronization

In case of objects that are not thread-safe a lock can be used.

Python 2.x

```
import thread,time
lock = thread.allocate_lock()
def MyPrint(sleepPeriod,name,count):
        global lock
        for i in range(0,count):
                lock.acquire()
                print (name+"=>"+str(i))
                lock.release()
                time.sleep(sleepPeriod)
thread.start_new_thread (MyPrint, (1,"Thread #1", 3))
thread.start_new_thread (MyPrint, (2,"Thread #2", 3))
thread.start_new_thread (MyPrint, (3,"Thread #3", 4))
time.sleep(10)
```

Output

```
Thread #1=>0
Thread #3=>0
Thread #2=>0
Thread #1=>1
Thread #2=>1
Thread #1=>2
Thread #3=>1
Thread #2=>2
Thread #3=>2
```

Threading and Synchronization

Locks can also be used with **with** statement (in this case the acquire and release are called in **__enter __** and **__exit__** code)

Python 2.x

```
import thread,time
lock = thread.allocate_lock()
def MyPrint(sleepPeriod,name,count):
        global lock
        for i in range(0,count):
                with lock:
                        print (name+"=>"+str(i))
                time.sleep(sleepPeriod)
thread.start_new_thread (MyPrint, (1,"Thread #1", 3))
thread.start_new_thread (MyPrint, (2,"Thread #2", 3))
thread.start_new_thread (MyPrint, (3,"Thread #3", 4))
time.sleep(10)
```

Output

```
Thread #1=>0
Thread #3=>0
Thread #2=>0
Thread #1=>1
Thread #2=>1
Thread #1=>2
Thread #3=>1
Thread #2=>2
Thread #3=>2
```

Threading and Synchronization

Locks can also be used to wait for a thread to finish.

Python 2.x

```
import thread,time
lock = thread.allocate_lock()
lock.acquire()
def MyPrint(sleepPeriod,name,count):
        global lock
        for i in range(0,count):
                print (name+"=>"+str(i))
                time.sleep(sleepPeriod)
        lock.release()
thread.start_new_thread (MyPrint, (1,"Thread #1", 3))
print ("Waiting for a thread to finish ...")
lock.acquire()
print ("Thread finished")
```

Output

```
Waiting for a thread to finish ...
Thread #1=>0
Thread #1=>1
Thread #1=>2
Thread finished
```

Threading and Synchronization

Locks can also be used to wait for a thread to finish.

Python 2.x

```
import thread,time
lock = thread.allocate_lock()
lock.acquire()
def MyPrint(sleepPeriod,name,count):
        global lock
        for i in range(0,count):
                print (name+"=>"+str(i))
                time.sleep(sleepPeriod)
        lock.release()
thread.start_new_thread (MyPrint, (1,"Thread #1", 3))
print ("Waiting for a thread to finish ...")
lock.acquire()
print ("Thread finished")
```

Step 1:
lock variable is acquired before any thread is started.

Threading and Synchronization

Locks can also be used to wait for a thread to finish.

Python 2.x

```
import thread,time
lock = thread.allocate_lock()
lock.acquire()
def MyPrint(sleepPeriod,name,count):
        global lock
        for i in range(0,count):
                print (name+"=>"+str(i))
                time.sleep(sleepPeriod)
        lock.release()
thread.start_new_thread (MyPrint, (1,"Thread #1", 3))
print ("Waiting for a thread to finish ...")
lock.acquire()
print ("Thread finished")
```

Step 2:
Main thread tries to acquire again the **lock** variable. As this variable was already acquired, the main thread will wait until **lock** variable is released.

Threading and Synchronization

Locks can also be used to wait for a thread to finish.

Python 2.x

```
import thread,time
lock = thread.allocate_lock()
lock.acquire()
def MyPrint(sleepPeriod,name,count):
        global lock
        for i in range(0,count):
                print (name+"=>"+str(i))
                time.sleep(sleepPeriod)
        lock.release()
thread.start_new_thread (MyPrint, (1,"Thread #1", 3))
print ("Waiting for a thread to finish ...")
lock.acquire()
print ("Thread finished")
```

Step 3:
When “Thread #1” is finished the **lock** variable is released. At that point the call to **lock.acquire** from the main thread will be executed and the script will continue.

Threading and Synchronization

Exceptions not caught in a different thread than the main thread will not stop the program.

Python 2.x

```
import thread,time
def MyPrint(sleepPeriod,name,count):
        global lock
        for i in range(-count,count):
                print (name+"=>"+str(10/i))
                time.sleep(sleepPeriod)
thread.start_new_thread (MyPrint, (1,"Threa
for i in range(0,10):
        print ("Main thread : "+str(i))
        time.sleep(1)
```

Output

```
Main thread : 0Thread #1=>-4
Thread #1=>-5Main thread : 1
Main thread : 2
Thread #1=>-10
Main thread : 3
Unhandled exception in thread started
Traceback (most recent call last):
ZeroDivisionError: integer division
or modulo by zero
Main thread : 4
Main thread : 5
Main thread : 6
Main thread : 7
Main thread : 8
Main thread : 9
```

Threading and Synchronization

Threading module provides high level functions for thread workers and synchronization.

It also provides a class **Thread** that can be used to derive thread based objects. When deriving from a **Thread** class two methods are usually implemented:

- run() ➔ code that will be executed when the thread starts
- __init__ ➔ thread constructor (it is important to call __init__ from thee base class before doing anything with the thread

Thread class has the following methods:

- **start()** ➔ starts the thread
- **join(***timeout***)** ➔ waits for the thread to finish
- getName/setName and name attribute ➔ indicate the name of the thread (if needed)
- **is_alive**() ➔ return true if the thread is alive

Threading and Synchronization

Using threading.Thread without subclassing

Python 2.x/3.x

```
import threading,time


def WaitSomeSeconds(seconds):
        time.sleep(seconds)

t = threading.Thread(target=WaitSomeSeconds, args = (5,))
t.start()
print("Wait for the thread to complete ...")
t.join()
```

Output

```
Wait for the thread to complete ...
```

The target function expect a touple with arguments. If that touple contains only one parameter, a ‘,’ must be added to specify a touple.

Threading and Synchronization

Using threading.Thread without subclassing

Python 2.x/3.x

```
import threading,time


def WaitSomeSeconds(seconds,x,y):
        time.sleep(seconds)
        print(x+y)

t = threading.Thread(target=WaitSomeSeconds, args = (5,10,20))
t.start()
print("Wait for the thread to complete ...")
t.join()
```

Output

```
Wait for the thread to complete ...
30
```

Threading and Synchronization

Subclassing threading.Thread. Thread code will be added in “run” method.

Python 2.x/3.x

```
import threading,time

class Mythread(threading.Thread):
        def __init__(self,seconds):
                threading.Thread.__init__(self)
                self.seconds = seconds
        def run(self):
                time.sleep(self.seconds)

t = Mythread(3)
t.start()
print("Wait for the thread to complete ...")
t.join()
```

Output

```
Wait for the thread to complete ...
```

Synchronization

The following synchronization object are available in **threading** module:

- lock
- rlock (reentrant lock)
- Condition objects
- Semaphore
- Event
- Timer
- Barrier

Synchronization (Lock)

Allows synchronized access to a resource.

Lock objects have two functions:

1. Python 3: Lock.**acquire**(blocking=True, timeout=-1) (timeout means how many seconds the Lock has to wait until it is acquired.
   Python 2: Lock.**acquire**(blocking=True)
   Lock.**acquire** returns true if the lock was acquired, false otherwise.
2. Lock.**release**() ➔ releases the lock. If called on an unlocked lock, an error will be raised.

Lock objects also support working with **with** keyword.

Synchronization (Lock)

Using Lock object (there is no guarantee that the number will be in order !!!)

Python 2.x/3.x

```
import threading,time
l = threading.Lock()
def ThreadFnc(lock,n_list,start):
        for i in range(0,10):
                lock.acquire()
                n_list+=[start+i]
                lock.release()
                time.sleep(1)
lst = []
t1 = threading.Thread(target=ThreadFnc, args=(l,lst,100))
t2 = threading.Thread(target=ThreadFnc, args=(l,lst,1000))
t1.start ()
t2.start ()
t1.join ()
t2.join ()
```

Python 2

```
[100, 1000, 101, 1001, 102, 1002, 103, 1003, 104,
1004, 105, 1005, 106, 1006, 107, 1007, 108, 1008,
109, 1009]
```

Python 3

```
[100, 1000, 1001, 101, 1002, 102, 1003, 103, 1004,
104, 1005, 105, 106, 1006, 107, 1007, 108, 1008, 109,
1009]
```

Synchronization (Lock)

Using Lock object with **with** keyword

Python 2.x/3.x

```
import threading,time
l = threading.Lock()
def ThreadFnc(lock,n_list,start):
        for i in range(0,10):
                with lock: n_list+=[start+i]
                time.sleep(1)
lst = []
t1 = threading.Thread(target=ThreadFnc, args=(l,lst,100))
t2 = threading.Thread(target=ThreadFnc, args=(l,lst,1000))
t1.start ()
t2.start ()
t1.join ()
t2.join ()
```

Synchronization (RLock)

Allows reentrant lock (the same thread can lock a resources multiple times).

RLock objects have two functions:

1. Python 3: Lock.**acquire**(blocking=True, timeout=-1) (timeout means how many seconds the Lock has to wait until it is acquired.
   Python 2: Lock.**acquire**(blocking=True)
   Lock.**acquire** returns true if the lock was acquired, false otherwise. If the lock was already acquire by the same thread, a counter is increased and **true** is returned.
2. Lock.**release**() ➔ decreases the counter. Once it reaches 0, the lock is unlocked.

RLock objects also support working with **with** keyword.

Within the same thread, be sure that the number of **acquire** queries is thee same as the number of **release** (otherwise you risk keeping the lock unlocked !!!)

Synchronization (RLock)

Python 2.x/3.x

```
import threading
l = threading.Lock()
def ThreadFnc1(lock):
        with lock: print("fnc_1_called")
def ThreadFnc2(lock):
        with lock:
                print("fnc_2_called")
                ThreadFnc1(lock)
t1 = threading.Thread(target=ThreadFnc1, args=(l,))
t2 = threading.Thread(target=ThreadFnc2, args=(l,))
t1.start ()
t2.start ()
t1.join ()
t2.join ()
```

Current program will never end. When ThreadFnc2 calls ThreadFnc1, the lock is already block and a dead-lock is produced.

Synchronization (RLock)

Python 2.x/3.x

```
import threading
l = threading.RLock()
def ThreadFnc1(lock):
        with lock: print("fnc_1_called")
def ThreadFnc2(lock):
        with lock:
                print("fnc_2_called")
                ThreadFnc1(lock)
t1 = threading.Thread(target=ThreadFnc1, args=(l,))
t2 = threading.Thread(target=ThreadFnc2, args=(l,))
t1.start ()
t2.start ()
t1.join ()
t2.join ()
```

If we replace Lock with RLock the same code will function as it should.

Synchronization (Condition object)

Provides a notification system to other systems based on a condition. It has the following methods:

- **acquire**
- **release**
- **wait**
- **wait_for (Python 3)**
- **notify**
- **notify_all**

Conditional objects also support working with **with** keyword.

Synchronization (Condition object)

Python 2.x/3.x

```
import threading,time
c = threading.Condition()
number = 0
def ThreadConsumer():
          global number,c
          with c:
                    if number==0: c.wait()
                    print("Consume: "+str(number))
                    number = 0
def ThreadProducer():
          global number,c
          with c:
                    time.sleep(2)
                    number = 5
                    c.notify()
t1 = threading.Thread(target=ThreadConsumer)
t2 = threading.Thread(target=ThreadProducer)
t1.start ()
t2.start ()
t1.join ()
t2.join ()
```

Output (after 2 seconds)

```
Consume: 5
```

Synchronization (Condition object)

Python 3.x

```
import threading,time
c = threading.Condition()
number = 0
def ThreadConsumer():
        global number,c
        with c:
                c.wait_for(lambda: number!=0)
                print("Consume: "+str(number))
                number = 0
def ThreadProducer():
        global number,c
        with c:
                time.sleep(2)
                number = 5
                c.notify()
t1 = threading.Thread(target=ThreadConsumer)
t2 = threading.Thread(target=ThreadProducer)
t1.start ()
t2.start ()
t1.join ()
t2.join ()
```

Output (after 2 seconds)

```
Consume: 5
```

Synchronization (Semaphores)

Provides access to a limited number of threads to a resource. It has the following functions:

- **acquire**
- **release**

Conditional objects also support working with **with** keyword.

Synchronization (Semaphores)

Python 2.x/3.x

```
import threading,time
s = threading.Semaphore(4)
def WorkerThread(id):
        global s
        with s:
                print("Thread-#"+str(id)+" enter")
                time.sleep(1)
                print("Thread-#"+str(id)+" exit")
t = []
for i in range(0,10):
        t += [threading.Thread(target=WorkerThread, args=(i,))]
for _th in t: _th.start ()
for _th in t: _th.join ()
```

Output

```
Thread-#0 enter
Thread-#1 enter
Thread-#2 enter
Thread-#3 enter
Thread-#3 exit
Thread-#2 exit
Thread-#0 exit
Thread-#4 enter
Thread-#1 exit
Thread-#5 enter
Thread-#6 enter
Thread-#7 enter
Thread-#6 exit
Thread-#5 exit
Thread-#4 exit
Thread-#8 enter
Thread-#9 enter
Thread-#7 exit
Thread-#8 exit
Thread-#9 exit
```

Synchronization (Timer)

Timer is an object deriver from Thread. It allows to run a code after a specific period of time. A timer also have a **cancel** method to stop the timer.

Python 2.x/3.x

```
import threading,time

def TimerFunction(mesaj):
        print (mesaj)

timer = threading.Timer(5,TimerFunction,("test after 5 seconds",))
timer.start()
timer.join()
print("Done")
```

Output

```
test after 5 seconds
Done
```

Synchronization (Event)

Event object provides a way to synchronize execution between two or more threads.

It has the following functions:

- set ➔ to signal the current state of the event
- clear ➔ to clear the current state of the event
- wait ➔ wait until the event is signaled (a call to **set** method was made)
- is_set ➔ to check if an event was signaled

Events can not be used with **with** keyword.

To synchronize two thread, two Events are usually used.

Synchronization (Event)

Python 2.x/3.x

```
import threading
e1 = threading.Event()
e2 = threading.Event()
e1.set()
def AddNumber(start,event1,event2,lista):
        for i in range(start,10,2):
                event1.wait()
                event1.clear()
                lista += [i]
                event2.set()
l = []
t1 = threading.Thread(target=AddNumber, args=(1,e1,e2,l))
t2 = threading.Thread(target=AddNumber, args=(2,e2,e1,l))
t1.start()
t2.start()
t1.join()
t2.join()
print (l)
```

Output

```
[0,1,2,3,4,5,6,7,8,9]
```

Synchronization (Barrier)

Provides a mechanism to wait for multiple threads to start at the same time.

It has the following functions:

- **wait** ➔ wait until the number if threads that need to pass a barrier is completed. Only then all threads are released and will continue their execution
- **Reset** ➔ resets the barrier
- **abort** ➔ aborts current barrier
- **parties** ➔ number of parties (threads) that has to pass the barrier

Barriers can not be used with **with** keyword.

Barriers are available only on Python 3.

Synchronization (Barrier)

Python 2.x/3.x

```
import threading,time
b = threading.Barrier(2)

def WorkerThread(b,id):
        b_id = b.wait()
        print("#"+str(id)+" pass the barier => "+str(b_id))
        time.sleep(2)
        print("#"+str(id)+" exit")

t = []
for i in range(0,10):
        t += [threading.Thread(target=WorkerThread, args=(i,))]
for _th in t: _th.start ()
for _th in t: _th.join ()
```

Synchronization (Barrier)

Python 2.x/3.x

```
import threading,time
b = threading.Barrier(2)

def WorkerThread(b,id):
        b_id = b.wait()
        print("#"+str(id)+" pass the barier => "+str(b_
        time.sleep(2)
        print("#"+str(id)+" exit")

t = []
for i in range(0,10):
        t += [threading.Thread(target=WorkerThread, arg
for _th in t: _th.start ()
for _th in t: _th.join ()
```

Output

```
#1 pass the barier => 1
#0 pass the barier => 0
#3 pass the barier => 1
#2 pass the barier => 0
#5 pass the barier => 1
#4 pass the barier => 0
#7 pass the barier => 1
#6 pass the barier => 0
#9 pass the barier => 1
#8 pass the barier => 0
#1 exit
#3 exit
#2 exit
#0 exit
#6 exit
#4 exit
#9 exit
#5 exit
#7 exit
#8 exit
```

Synchronization (Barrier)

Python 2.x/3.x

```
import threading,time
b = threading.Barrier(2)

def WorkerThread(b,id):
        b_id = b.wait()
        print("#"+str(id)+" pas
        time.sleep(2)
        print("#"+str(id)+" exi

t = []
for i in range(0,10):
        t += [threading.Thread(t
for _th in t: _th.start ()
for _th in t: _th.join ()
```

Each barrier waits for 2 thread. The **b_id** parameter indicates the id of a thread inside a barrier. The call to **wait** exits only when all threads that need to pass the barrier are present (in this case from 2 to 2 threads).

Output

```
#1 pass the barier => 1
#0 pass the barier => 0
#3 pass the barier => 1
#2 pass the barier => 0
#5 pass the barier => 1
#4 pass the barier => 0
#7 pass the barier => 1
#6 pass the barier => 0
#9 pass the barier => 1
#8 pass the barier => 0
#1 exit
#3 exit
#2 exit
#0 exit
#6 exit
#4 exit
#9 exit
#5 exit
#7 exit
#8 exit
```

Synchronization (Barrier)

Python 2.x/3.x

```
import threading,time
b = threading.Barrier(3)

def WorkerThread(b,id):
        b_id = b.wait()
        print("#"+str(id)+" pass the barier => "+str(b_
        time.sleep(2)
        print("#"+str(id)+" exit")

t = []
for i in range(0,10):
        t += [threading.Thread(target=WorkerThread, ar
for _th in t: _th.start ()
for _th in t: _th.join ()
```

Threads will be group in groups of 3. As there are 10 threads, thread no. 10 will never end (b.wait will wait until two more threads will enter in the barrier).