5CM510 – Network System Development
Lecture 4: Threading and Python Network programming

derby.ac.uk

Sensitivity: Internal

Learning outcomes

1. Understanding Asyncio
2. Using Threading Module
3. Python’s Multiprocessing Module

Asyncio, Threading and Multiprocessing

- **Python offers diverse paradigms for concurrent and parallel execution:**
- **Asyncio for asynchronous programming, Threading for concurrent execution, and Multiprocessing for parallel execution.**
- **Understanding their nuances, especially in the context of real-world applications, is crucial for writing efficient Python applications.**

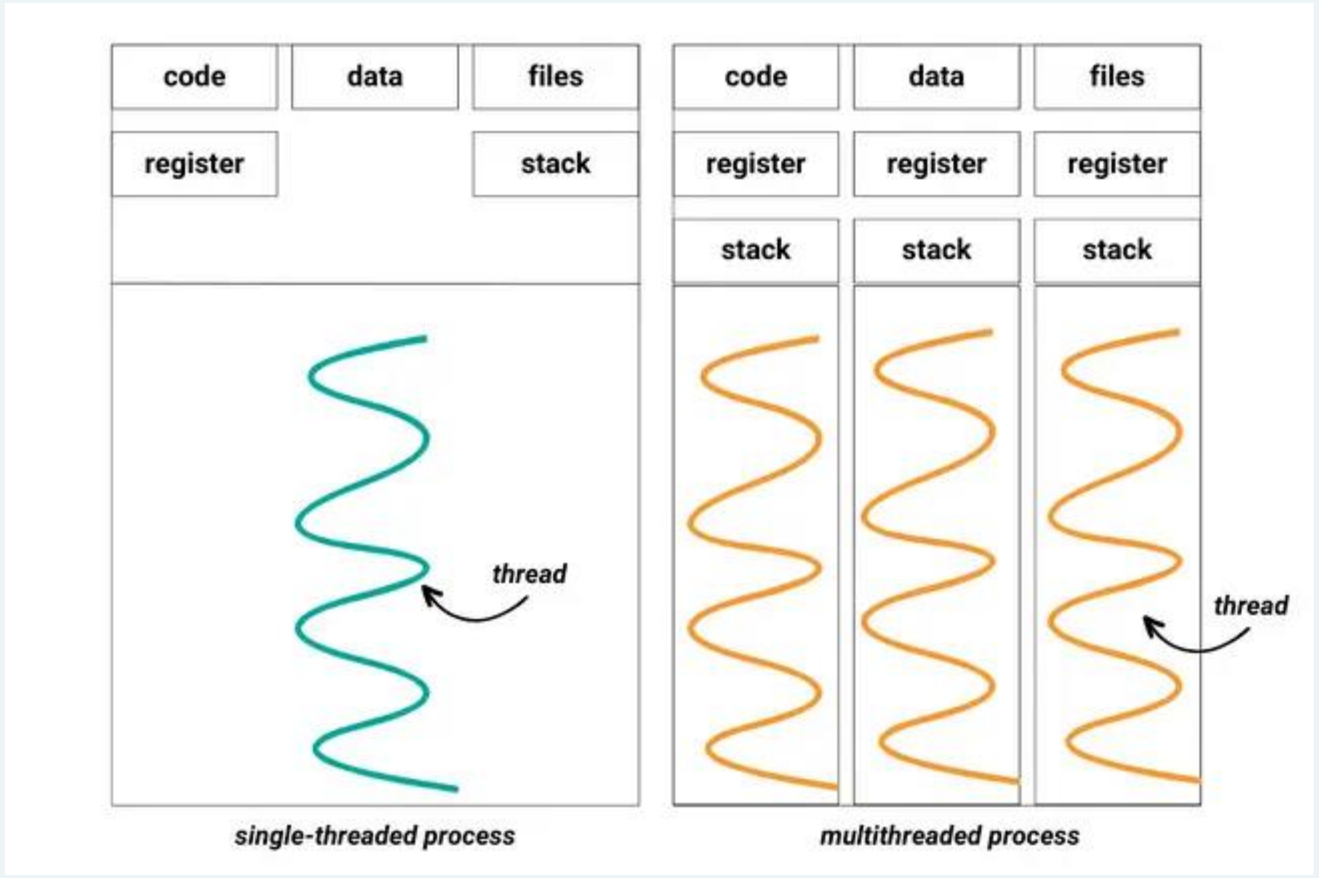
Asyncio

- Asyncio is a Python library designed for writing single-threaded concurrent code using coroutines.
- It excels in situations involving high-level structured network code or when handling multiple I/O-bound tasks simultaneously.
- The example (next slide) demonstrates Asyncio’s ability to handle multiple web requests concurrently, improving efficiency in I/O-bound tasks.

Asyncio: Example

```
import asyncio
import aiohttp

async def fetch_url(url):
    async with aiohttp.ClientSession() as session:
        async with session.get(url) as response:
            return await response.text()

async def main():
    urls = ["http://example.com", "http://example.org"]
    tasks = [fetch_url(url) for url in urls]
    results = await asyncio.gather(*tasks)
    return results

asyncio.run(main())
```

Single vs Multi-threaded process



Threading

- Threading allows for the execution of multiple threads in a single process.
- Due to the Global Interpreter Lock (GIL), Python threads don't execute bytecode in true parallelism but are useful for I/O-bound tasks.
- Global Interpreter Lock (GIL): A mutex that protects access to Python objects, preventing simultaneous execution of Python bytecode by multiple threads.
- I/O-Bound Efficiency: Threading is beneficial for tasks waiting for I/O operations, as threads can release the GIL during these operations, allowing others to run.
- The threading example uses threading for network operations, where each thread manages a different download task.

Threading: Code Example

```
import threading
import requests

def download_file(url):
    response = requests.get(url)
    print(f"Downloaded {url}")

urls = ["http://example.com/file1", "http://example.com/file2"]
threads = [threading.Thread(target=download_file, args=(url,)) for url in urls]

for thread in threads:
    thread.start()
for thread in threads:
    thread.join()
```

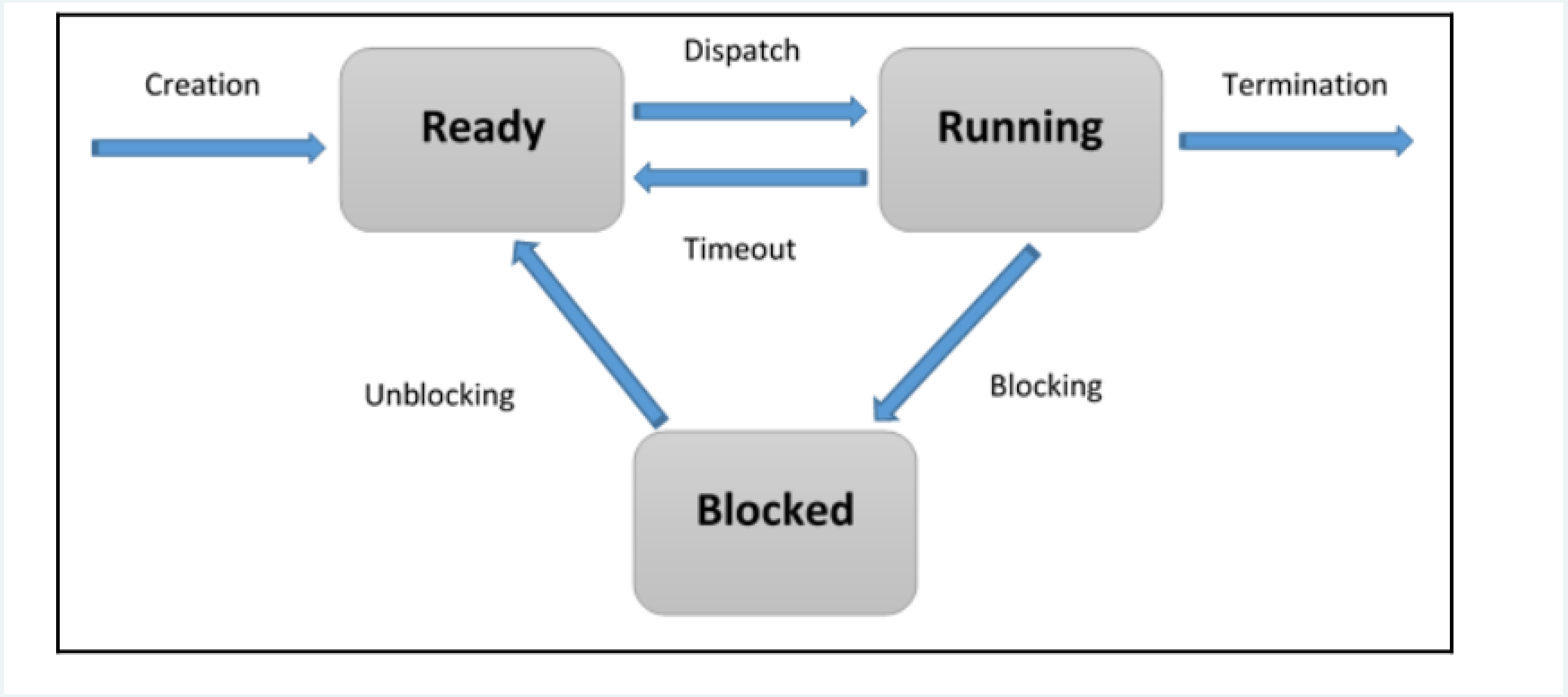
What is a Thread?

- *A thread is an independent execution flow that can be executed in parallel and concurrently with other threads in the system.*
- *Multiple threads can share data and resources, taking advantage of the so-called space of shared information. The specific implementation of threads and processes depends on the OS on which you plan to run the application, but, in general, it can be stated that a thread is contained inside a process and that different threads in the same process conditions share some resources. In contrast to this, different processes do not share their own resources with other processes.*
- *A thread is composed of three elements: program counters, registers, and stack. Shared resources with other threads of the same process essentially include data and OS resources.*
- *Moreover, threads have their own state of execution, namely, thread state, and can be synchronized with other threads.*

What is a Thread?

- *A thread state can be ready, running, or blocked: When a thread is created, it enters the Ready state.*
- *A thread is scheduled for execution by the OS (or by the runtime support system) and, when its turn arrives, it begins execution by going into the Running state.*
- *The thread can wait for a condition to occur, passing from the Running state to the Blocked state. Once the locked condition is terminated, the Blocked thread returns to the Ready state:*

Thread life cycle

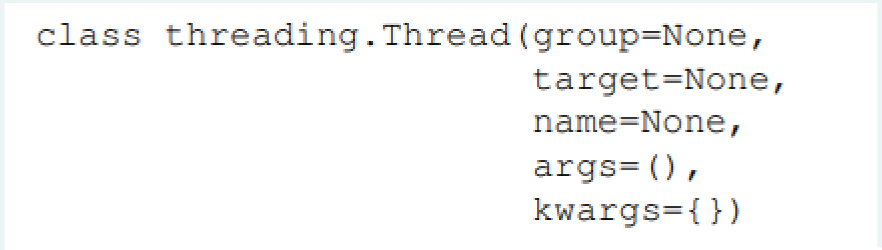


Python Threading module

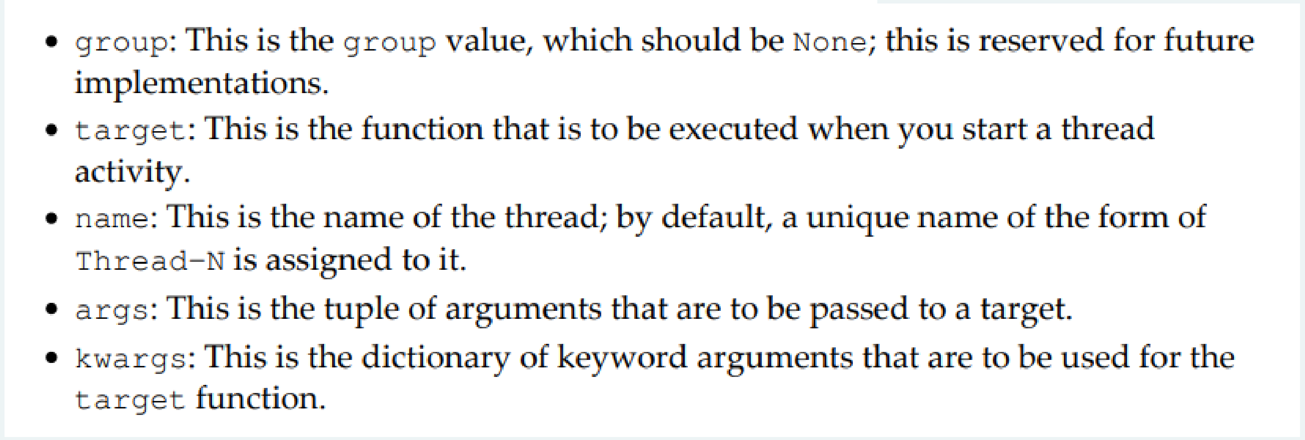
- Python manages threads with the threading module provided by the Python standard library. This module provides some very interesting features that make the threading-based approach a whole lot easier; in fact, the threading module provides several synchronization mechanisms that are very simple to implement.
- The major components of the threading module are as follows:
  - The thread object
  - The lock object
  - The RLock object
  - The semaphore object
  - The condition object
  - The event object

Defining a Thread

- The simplest way to use a thread is to instantiate it with a target function and then call the start method to let it begin the job.
- The Python threading module provides a Thread class that is used to run processes and functions in a different thread:



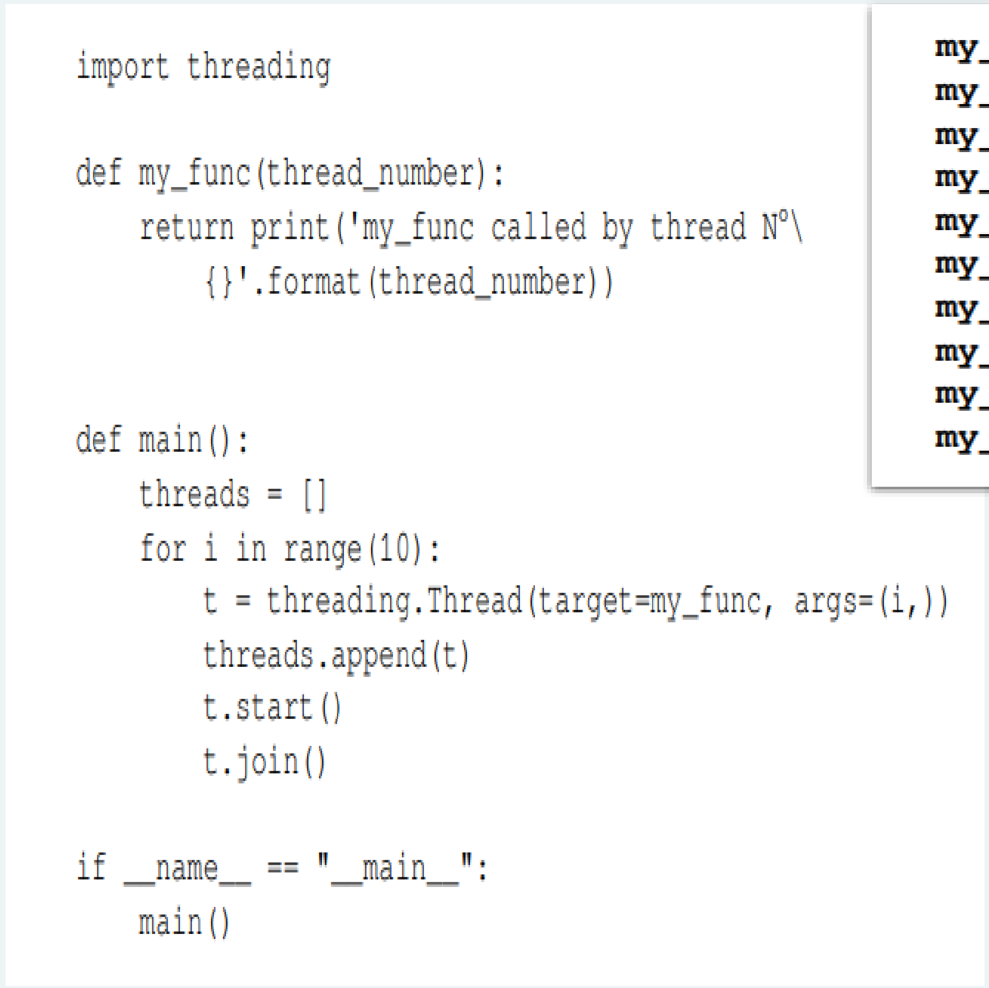
```
class threading.Thread(group=None,
                       target=None,
                       name=None,
                       args=(),
                       kwargs={})
```



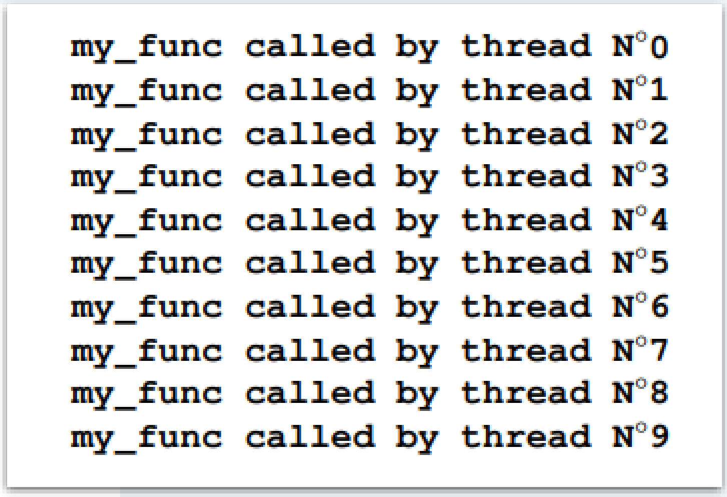
- `group`: This is the `group` value, which should be `None`; this is reserved for future implementations.
- `target`: This is the function that is to be executed when you start a thread activity.
- `name`: This is the name of the thread; by default, a unique name of the form of `Thread-N` is assigned to it.
- `args`: This is the tuple of arguments that are to be passed to a target.
- `kwargs`: This is the dictionary of keyword arguments that are to be used for the `target` function.

Executing multiple threads

- Import the threading module by using the following Python command:
  - import threading
- In the main program, a Thread object is instantiated with a target function called my_func. Then, an argument to the function that will be included in the output message is passed:
- t = threading.Thread(target=function , args=(i,))
- The thread does not start running until the start method is called, and the join method makes the calling thread and waits until the thread has finished the execution, as follows:



```
import threading

def my_func(thread_number):
    return print('my_func called by thread N°\
        {}'.format(thread_number))



def main():
    threads = []
    for i in range(10):
        t = threading.Thread(target=my_func, args=(i,))
        threads.append(t)
        t.start()
        t.join()

if __name__ == "__main__":
    main()
```



```
my_func called by thread N°0
my_func called by thread N°1
my_func called by thread N°2
my_func called by thread N°3
my_func called by thread N°4
my_func called by thread N°5
my_func called by thread N°6
my_func called by thread N°7
my_func called by thread N°8
my_func called by thread N°9
```

Thread synchronization with a Lock

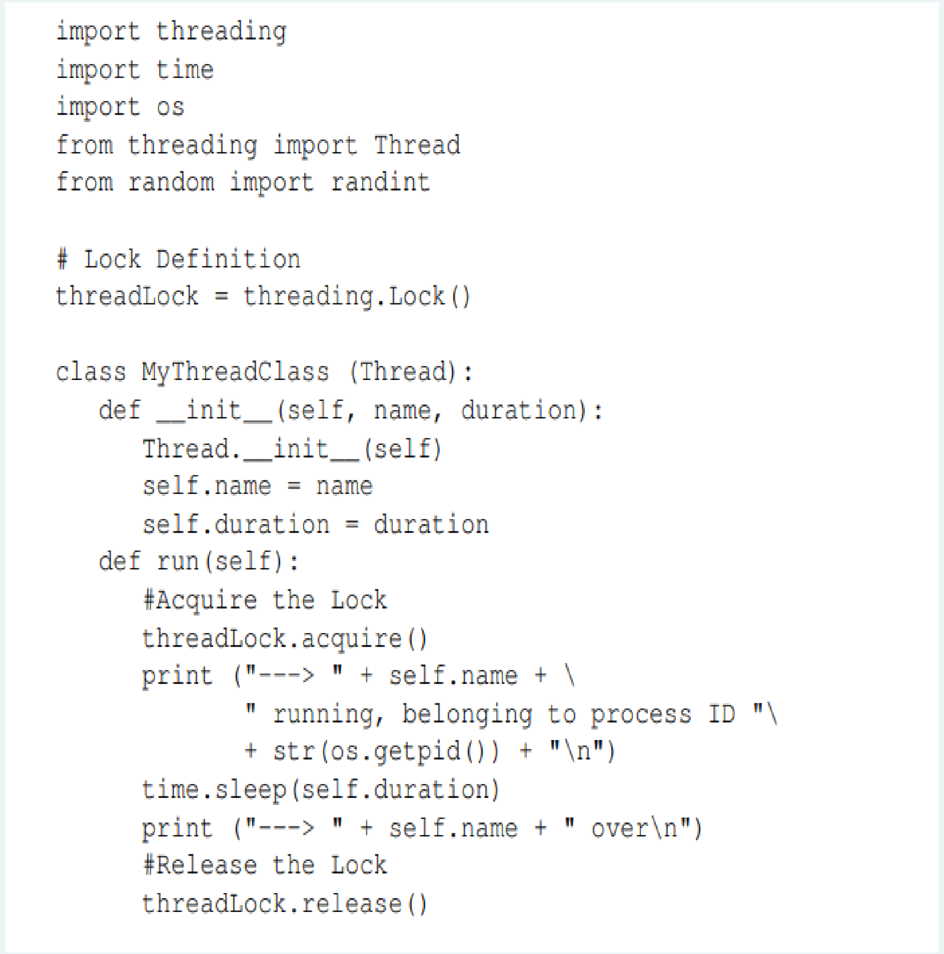
- A lock is nothing more than an object that is typically accessible by multiple threads, which a thread must possess before it can proceed to the execution of a protected section of a program.

- These locks are created by executing the Lock() method, which is defined in the threading module. Once the lock has been created, we can use two methods that allow us to synchronize the execution of two (or more) threads: the acquire() method to acquire the lock control, and the release() method to release it.

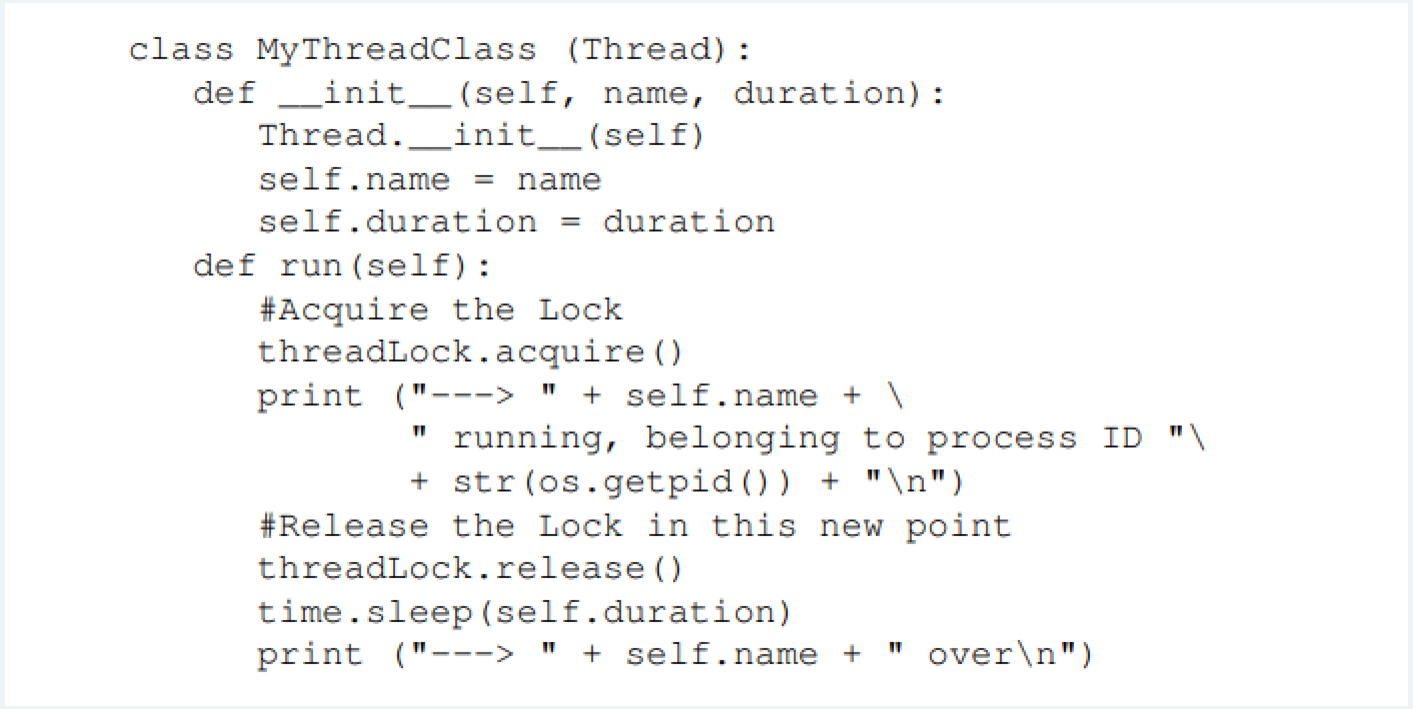
- The acquire() method accepts an optional parameter that, if not specified or set to True, forces the thread to suspend its execution until the lock is released and can then be acquired.

- If, on the other hand, the acquire() method is executed with an argument equal to False, then it immediately returns a Boolean result, which is True if the lock has been acquired, or False otherwise.

Introducing the acquire() and release() methods

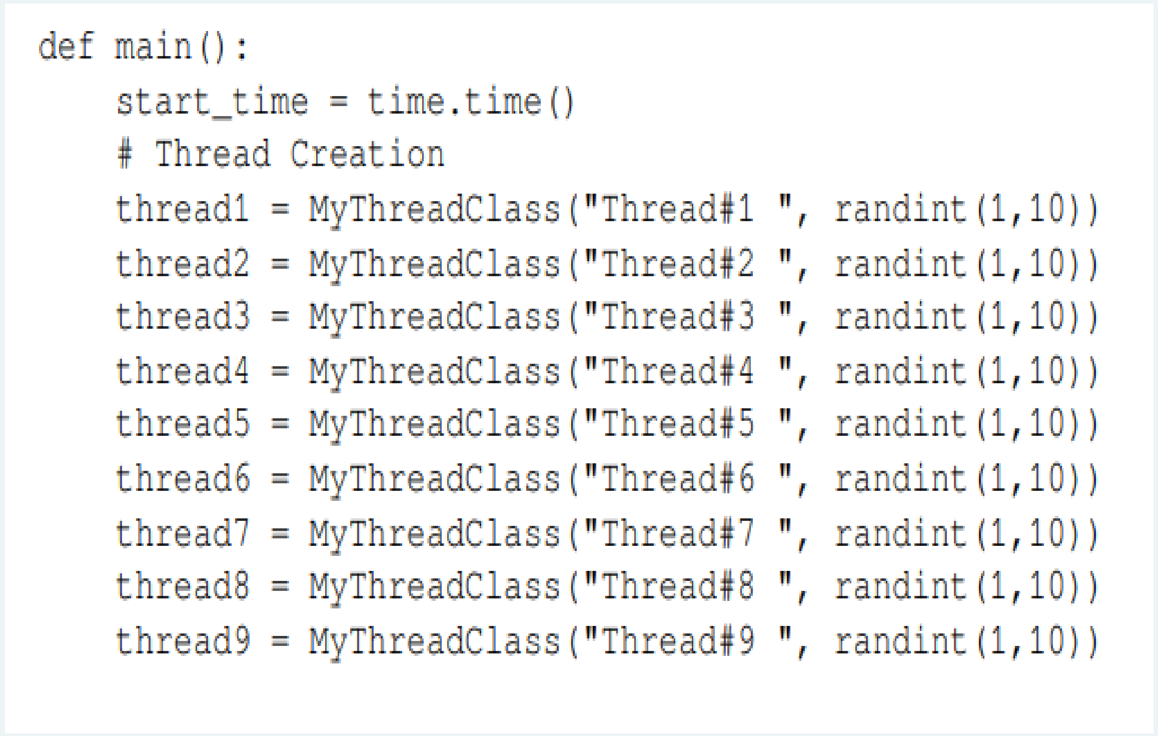


```
import threading
import time
import os
from threading import Thread
from random import randint

# Lock Definition
threadLock = threading.Lock()

class MyThreadClass (Thread):
   def __init__(self, name, duration):
      Thread.__init__(self)
      self.name = name
      self.duration = duration
   def run(self):
      #Acquire the Lock
      threadLock.acquire()
      print ("---> " + self.name + \
             " running, belonging to process ID "\
             + str(os.getpid()) + "\n")
      time.sleep(self.duration)
      print ("---> " + self.name + " over\n")
      #Release the Lock
      threadLock.release()
```

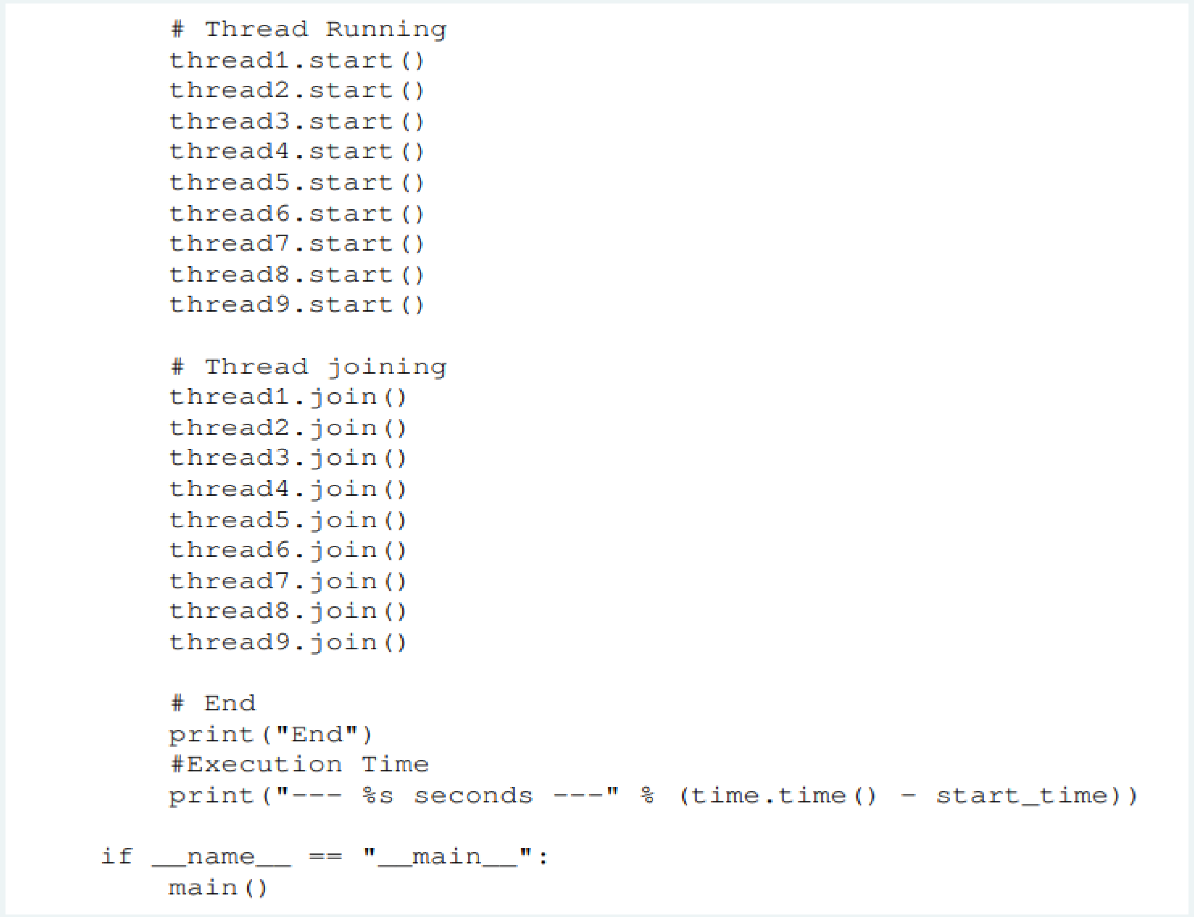


```
class MyThreadClass (Thread):
   def __init__(self, name, duration):
      Thread.__init__(self)
      self.name = name
      self.duration = duration
   def run(self):
      #Acquire the Lock
      threadLock.acquire()
      print ("---> " + self.name + \
             " running, belonging to process ID "\
             + str(os.getpid()) + "\n")
      #Release the Lock in this new point
      threadLock.release()
      time.sleep(self.duration)
      print ("---> " + self.name + " over\n")
```

Introducing the acquire() and release() methods

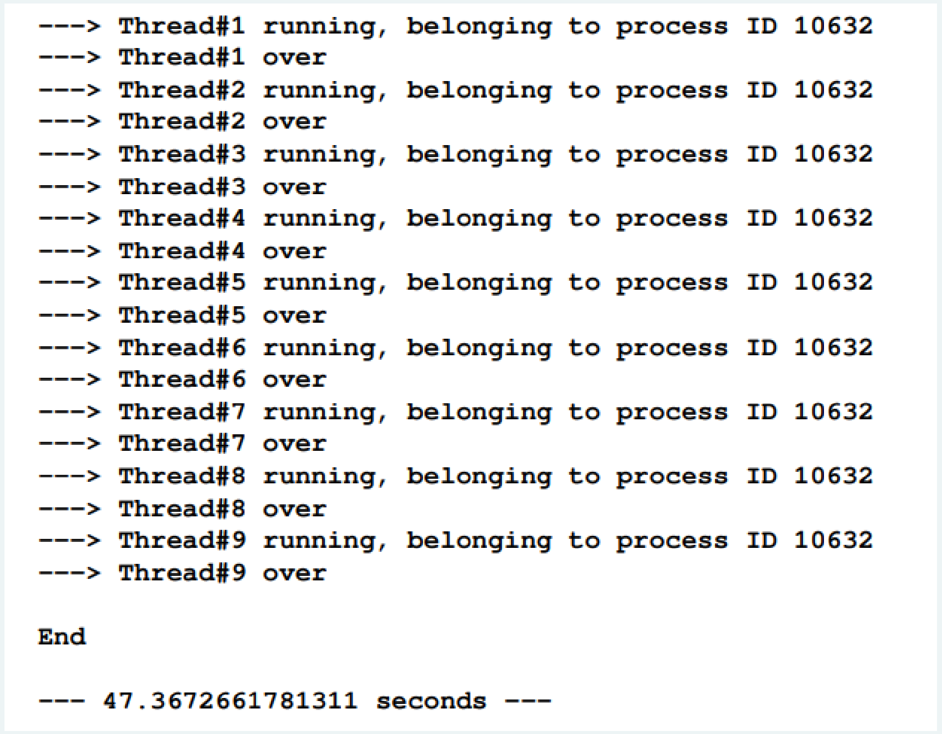


```
def main():
    start_time = time.time()
    # Thread Creation
    thread1 = MyThreadClass("Thread#1 ", randint(1,10))
    thread2 = MyThreadClass("Thread#2 ", randint(1,10))
    thread3 = MyThreadClass("Thread#3 ", randint(1,10))
    thread4 = MyThreadClass("Thread#4 ", randint(1,10))
    thread5 = MyThreadClass("Thread#5 ", randint(1,10))
    thread6 = MyThreadClass("Thread#6 ", randint(1,10))
    thread7 = MyThreadClass("Thread#7 ", randint(1,10))
    thread8 = MyThreadClass("Thread#8 ", randint(1,10))
    thread9 = MyThreadClass("Thread#9 ", randint(1,10))
```

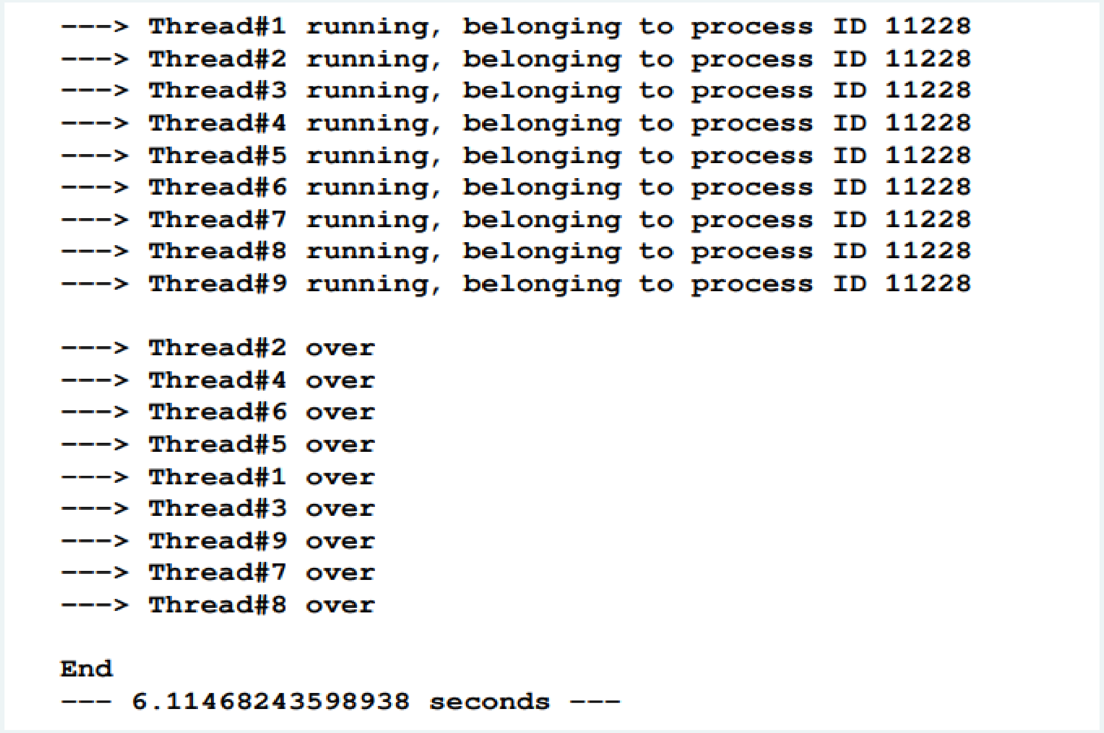


```
    # Thread Running
    thread1.start()
    thread2.start()
    thread3.start()
    thread4.start()
    thread5.start()
    thread6.start()
    thread7.start()
    thread8.start()
    thread9.start()

    # Thread joining
    thread1.join()
    thread2.join()
    thread3.join()
    thread4.join()
    thread5.join()
    thread6.join()
    thread7.join()
    thread8.join()
    thread9.join()

    # End
    print("End")
    #Execution Time
    print("--- %s seconds ---" % (time.time() - start_time))

if __name__ == "__main__":
    main()
```

Introducing the acquire() and release() methods



```
---> Thread#1 running, belonging to process ID 10632
---> Thread#1 over
---> Thread#2 running, belonging to process ID 10632
---> Thread#2 over
---> Thread#3 running, belonging to process ID 10632
---> Thread#3 over
---> Thread#4 running, belonging to process ID 10632
---> Thread#4 over
---> Thread#5 running, belonging to process ID 10632
---> Thread#5 over
---> Thread#6 running, belonging to process ID 10632
---> Thread#6 over
---> Thread#7 running, belonging to process ID 10632
---> Thread#7 over
---> Thread#8 running, belonging to process ID 10632
---> Thread#8 over
---> Thread#9 running, belonging to process ID 10632
---> Thread#9 over

End

--- 47.3672661781311 seconds ---
```



```
---> Thread#1 running, belonging to process ID 11228
---> Thread#2 running, belonging to process ID 11228
---> Thread#3 running, belonging to process ID 11228
---> Thread#4 running, belonging to process ID 11228
---> Thread#5 running, belonging to process ID 11228
---> Thread#6 running, belonging to process ID 11228
---> Thread#7 running, belonging to process ID 11228
---> Thread#8 running, belonging to process ID 11228
---> Thread#9 running, belonging to process ID 11228

---> Thread#2 over
---> Thread#4 over
---> Thread#6 over
---> Thread#5 over
---> Thread#1 over
---> Thread#3 over
---> Thread#9 over
---> Thread#7 over
---> Thread#8 over

End
--- 6.11468243598938 seconds ---
```

Thread synchronization with a RLock

- A reentrant lock, or simply an RLock, is a synchronization primitive that can be acquired multiple times by the same thread.
- It uses the concept of the proprietary thread. This means that in the locked state, some threads own the lock, while in the unlocked state, the lock is not owned by any thread.
- An RLock is implemented through the threading.RLock() class.
- It provides the acquire() and release() methods that have the same syntax as the threading.Lock() class.
- An RLock block can be acquired multiple times by the same thread. Other threads will not be able to acquire the RLock block until the thread that owns it has made a release() call for every previous acquire() call. Indeed, the RLock block must be released, but only by the thread that acquired it.
- The next example demonstrates how to manage threads through the RLock() mechanism.

Implementing RLock

- We introduced the Box class, which provides the add() and remove() methods
- Access the execute() method to perform the action to add or delete an item, respectively.

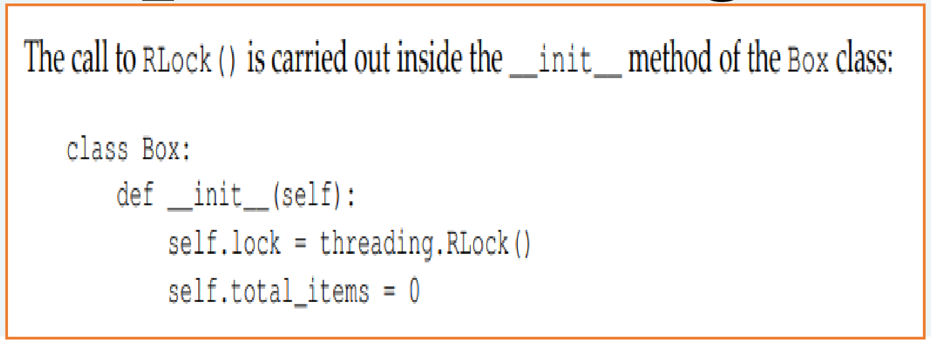
```
import threading
import time
import random

class Box:
    def __init__(self):
        self.lock = threading.RLock()
        self.total_items = 0

    def execute(self, value):
        with self.lock:
            self.total_items += value
```

```
def add(self):
    with self.lock:
        self.execute(1)

def remove(self):
    with self.lock:
        self.execute(-1)
```

- The functions below are called by the two threads. They have the box class and the total number of items to add or to remove as parameters:

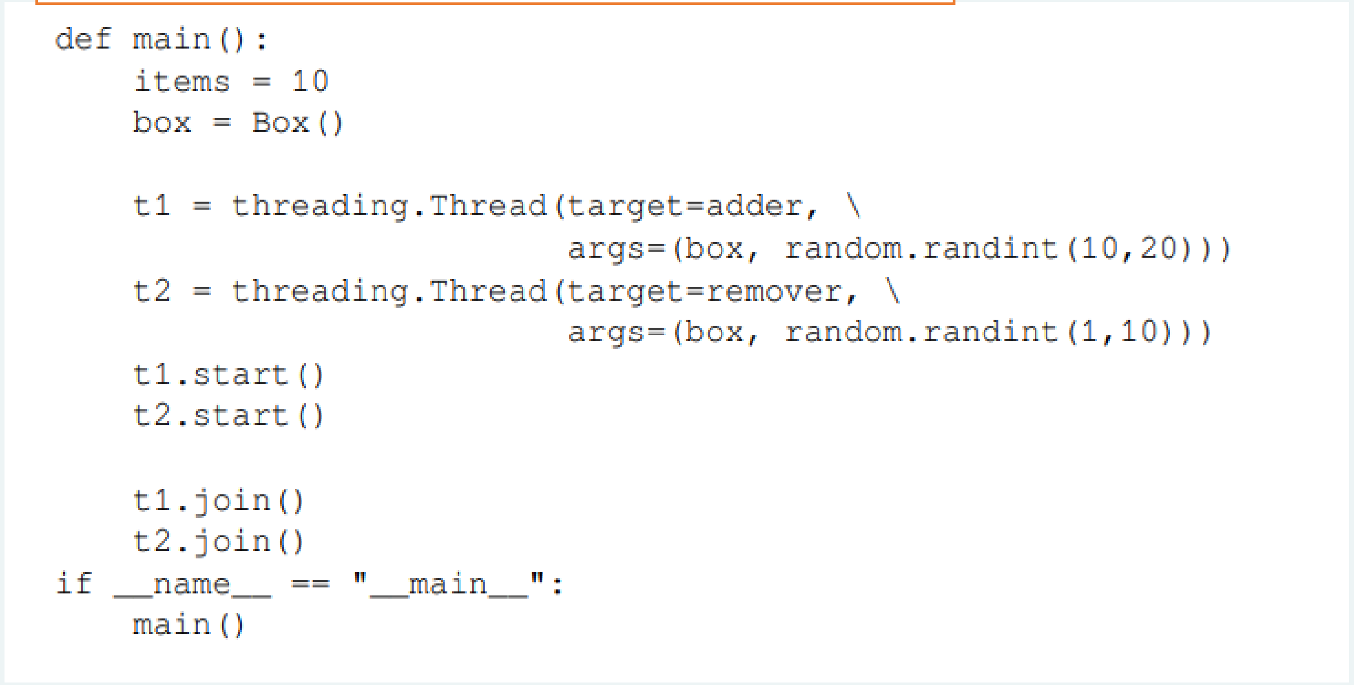
```
def adder(box, items):
    print("N° {} items to ADD \n".format(items))
    while items:
        box.add()
        time.sleep(1)
        items -= 1
        print("ADDED one item -->{} item to ADD \n".format(items))

def remover(box, items):
    print("N° {} items to REMOVE\n".format(items))
    while items:
        box.remove()
        time.sleep(1)
        items -= 1
        print("REMOVED one item -->{} item to REMOVE\
            \n".format(items))
```

Implementing RLock



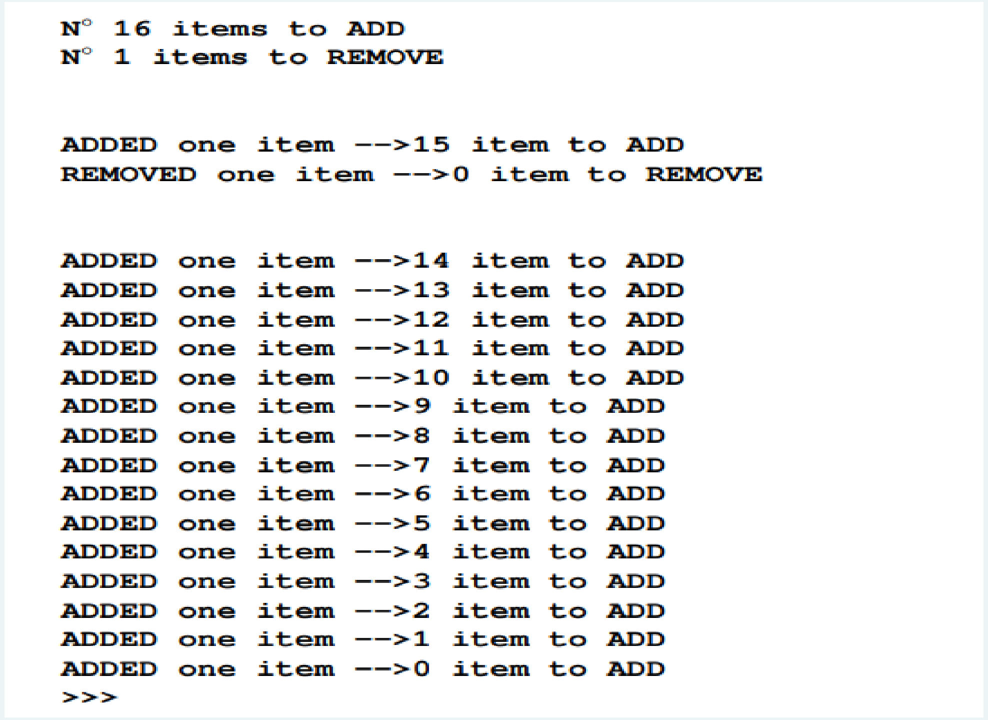
The call to RLock() is carried out inside the __init__ method of the Box class:

```
class Box:
    def __init__(self):
        self.lock = threading.RLock()
        self.total_items = 0
```



```
def main():
    items = 10
    box = Box()

    t1 = threading.Thread(target=adder, \
                          args=(box, random.randint(10,20)))
    t2 = threading.Thread(target=remover, \
                          args=(box, random.randint(1,10)))
    t1.start()
    t2.start()

    t1.join()
    t2.join()
if __name__ == "__main__":
    main()
```

- The total number of items to add or to remove from the box is set. As you can see, these two numbers will be different.
- The execution ends when both the adder and remover methods accomplish their tasks:



```
N° 16 items to ADD
N° 1 items to REMOVE


ADDED one item -->15 item to ADD
REMOVED one item -->0 item to REMOVE


ADDED one item -->14 item to ADD
ADDED one item -->13 item to ADD
ADDED one item -->12 item to ADD
ADDED one item -->11 item to ADD
ADDED one item -->10 item to ADD
ADDED one item -->9 item to ADD
ADDED one item -->8 item to ADD
ADDED one item -->7 item to ADD
ADDED one item -->6 item to ADD
ADDED one item -->5 item to ADD
ADDED one item -->4 item to ADD
ADDED one item -->3 item to ADD
ADDED one item -->2 item to ADD
ADDED one item -->1 item to ADD
ADDED one item -->0 item to ADD
>>>
```

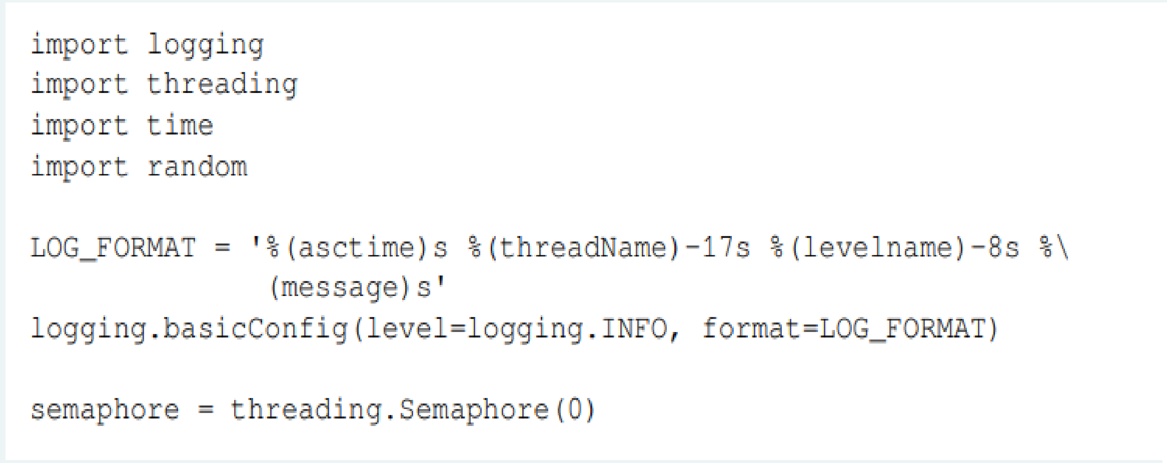
Thread synchronization with a semaphore

- The operation of a semaphore is based on two functions: acquire() and release(), as
- explained here:
- Whenever a thread wants to access a given or a resource that is associated with a semaphore, it must invoke the acquire() operation, which decreases the internal variable of the semaphore and allows access to the resource if the value of this variable appears to be non-negative.
- If the value is negative, then the thread will be suspended and the release of the resource by another thread will be placed on hold.
- Having finished using shared resources, the thread frees resources through the release() instruction. In this way, the internal variable of the semaphore is increased, allowing, for a waiting thread (if any), the opportunity to access the newly freed resource.
- The semaphore is one of the oldest synchronization primitives in the history of computer science, invented by the early **Dutch computer scientist Edsger W. Dijkstra**.
- The following example shows how to synchronize threads through a semaphore.

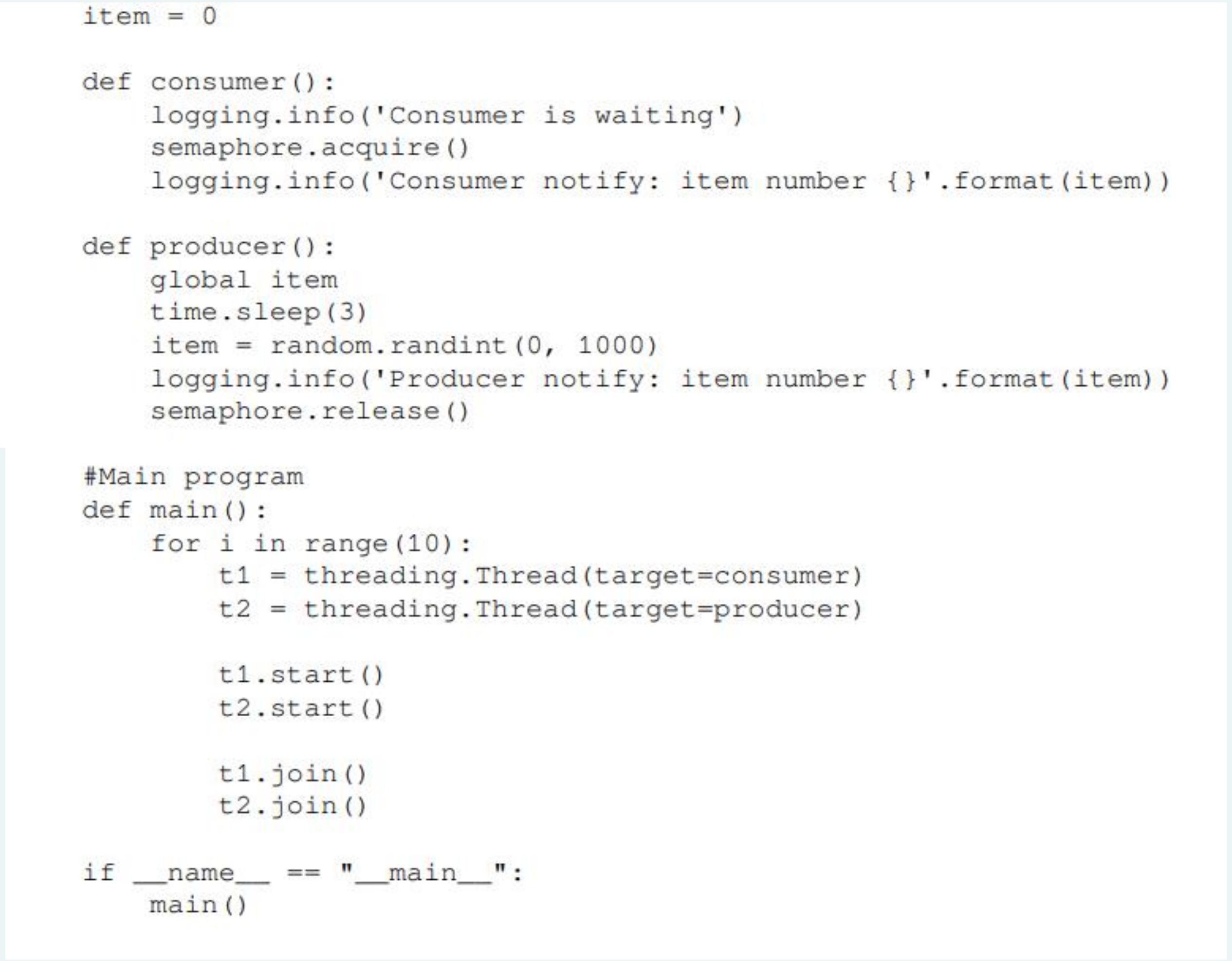
Thread synchronization with a semaphore

- By initializing a semaphore to 0, we obtain a so-called semaphore event whose sole purpose is to synchronize the computation of two or more threads. Here, a thread must make use of data or common resources simultaneously:
    - *semaphore = threading.Semaphore(0)*
- This operation is very similar to that described in the lock mechanism of the lock. The producer() thread creates the item and, after that, it frees the resource by calling the release() method:
    - *semaphore.release()*
- Similarly, the consumer() thread acquires the data by the acquire() method. If the semaphore's counter is equal to 0, then it blocks the condition's acquire() method until it gets notified by a different thread. If the semaphore's counter is greater than 0, then it decrements the value. When the producer creates an item, it releases the semaphore, and then the consumer acquires it and consumes the shared resource:
    - *semaphore.acquire()*

Implementing semaphore

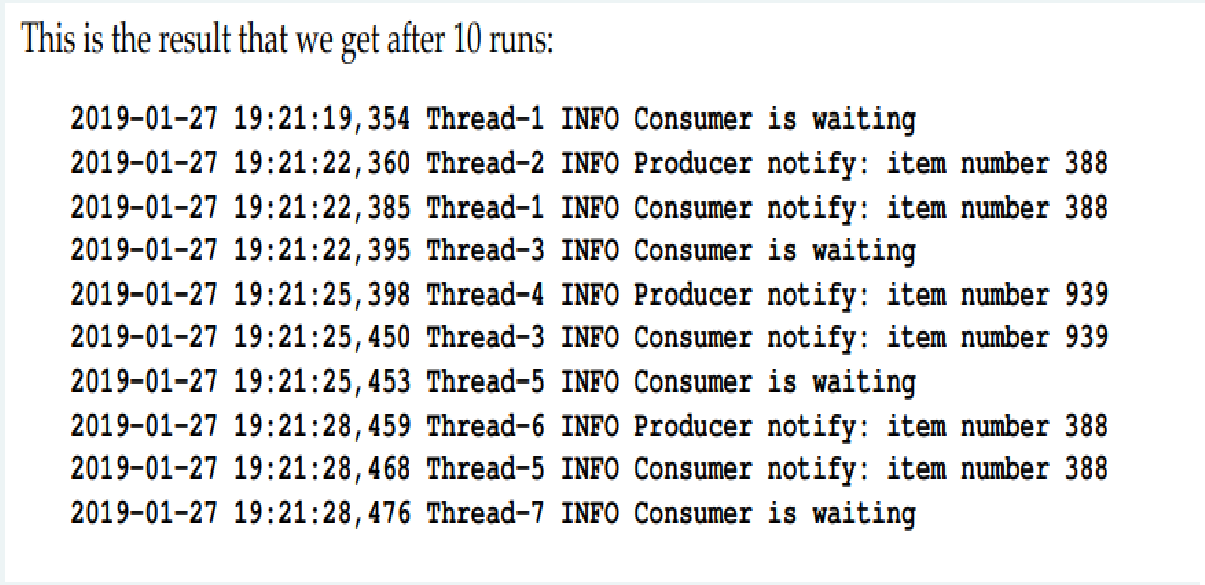


```
import logging
import threading
import time
import random

LOG_FORMAT = '%(asctime)s %(threadName)-17s %(levelname)-8s %\
              (message)s'
logging.basicConfig(level=logging.INFO, format=LOG_FORMAT)

semaphore = threading.Semaphore(0)
```



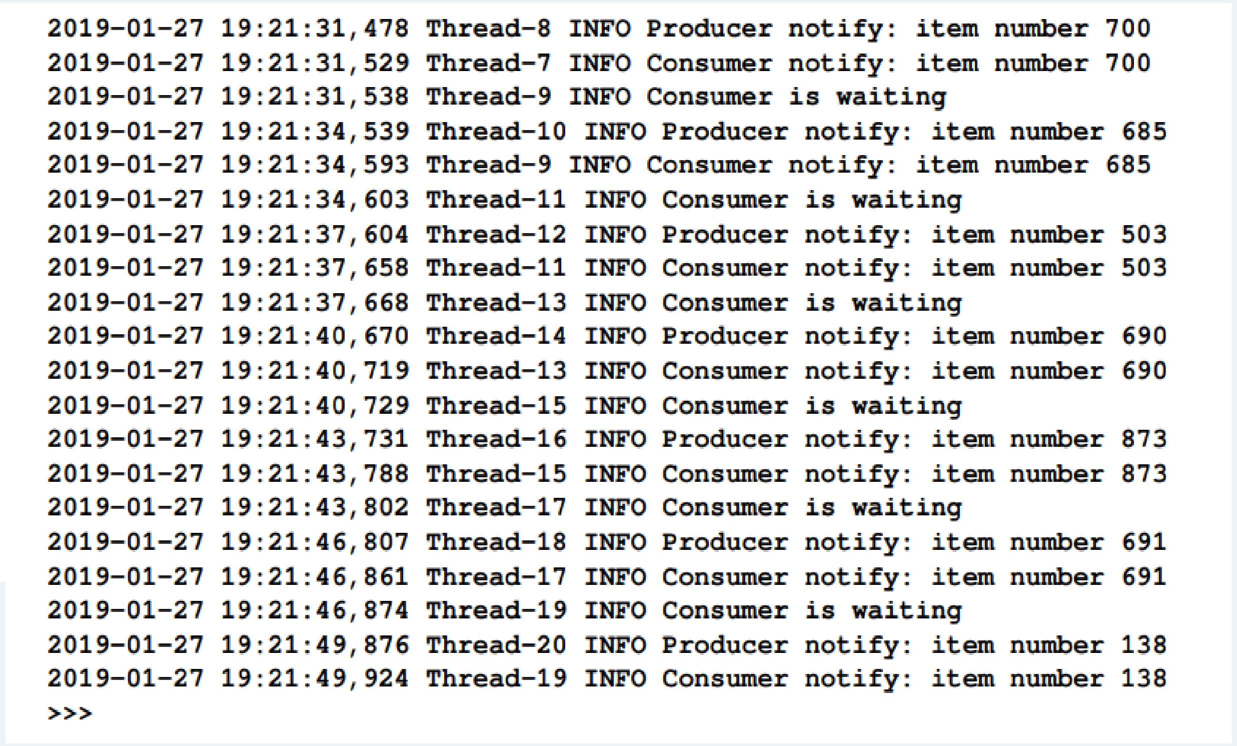
```
item = 0

def consumer():
    logging.info('Consumer is waiting')
    semaphore.acquire()
    logging.info('Consumer notify: item number {}'.format(item))

def producer():
    global item
    time.sleep(3)
    item = random.randint(0, 1000)
    logging.info('Producer notify: item number {}'.format(item))
    semaphore.release()

#Main program
def main():
    for i in range(10):
        t1 = threading.Thread(target=consumer)
        t2 = threading.Thread(target=producer)

        t1.start()
        t2.start()

        t1.join()
        t2.join()

if __name__ == "__main__":
    main()
```

Implementing semaphore



This is the result that we get after 10 runs:

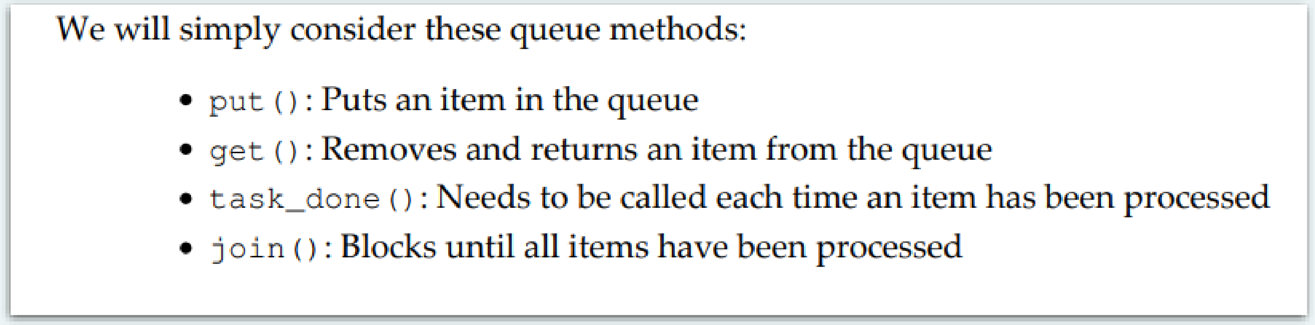
```
2019-01-27 19:21:19,354 Thread-1 INFO Consumer is waiting
2019-01-27 19:21:22,360 Thread-2 INFO Producer notify: item number 388
2019-01-27 19:21:22,385 Thread-1 INFO Consumer notify: item number 388
2019-01-27 19:21:22,395 Thread-3 INFO Consumer is waiting
2019-01-27 19:21:25,398 Thread-4 INFO Producer notify: item number 939
2019-01-27 19:21:25,450 Thread-3 INFO Consumer notify: item number 939
2019-01-27 19:21:25,453 Thread-5 INFO Consumer is waiting
2019-01-27 19:21:28,459 Thread-6 INFO Producer notify: item number 388
2019-01-27 19:21:28,468 Thread-5 INFO Consumer notify: item number 388
2019-01-27 19:21:28,476 Thread-7 INFO Consumer is waiting
```



```
2019-01-27 19:21:31,478 Thread-8 INFO Producer notify: item number 700
2019-01-27 19:21:31,529 Thread-7 INFO Consumer notify: item number 700
2019-01-27 19:21:31,538 Thread-9 INFO Consumer is waiting
2019-01-27 19:21:34,539 Thread-10 INFO Producer notify: item number 685
2019-01-27 19:21:34,593 Thread-9 INFO Consumer notify: item number 685
2019-01-27 19:21:34,603 Thread-11 INFO Consumer is waiting
2019-01-27 19:21:37,604 Thread-12 INFO Producer notify: item number 503
2019-01-27 19:21:37,658 Thread-11 INFO Consumer notify: item number 503
2019-01-27 19:21:37,668 Thread-13 INFO Consumer is waiting
2019-01-27 19:21:40,670 Thread-14 INFO Producer notify: item number 690
2019-01-27 19:21:40,719 Thread-13 INFO Consumer notify: item number 690
2019-01-27 19:21:40,729 Thread-15 INFO Consumer is waiting
2019-01-27 19:21:43,731 Thread-16 INFO Producer notify: item number 873
2019-01-27 19:21:43,788 Thread-15 INFO Consumer notify: item number 873
2019-01-27 19:21:43,802 Thread-17 INFO Consumer is waiting
2019-01-27 19:21:46,807 Thread-18 INFO Producer notify: item number 691
2019-01-27 19:21:46,861 Thread-17 INFO Consumer notify: item number 691
2019-01-27 19:21:46,874 Thread-19 INFO Consumer is waiting
2019-01-27 19:21:49,876 Thread-20 INFO Producer notify: item number 138
2019-01-27 19:21:49,924 Thread-19 INFO Consumer notify: item number 138
>>>
```

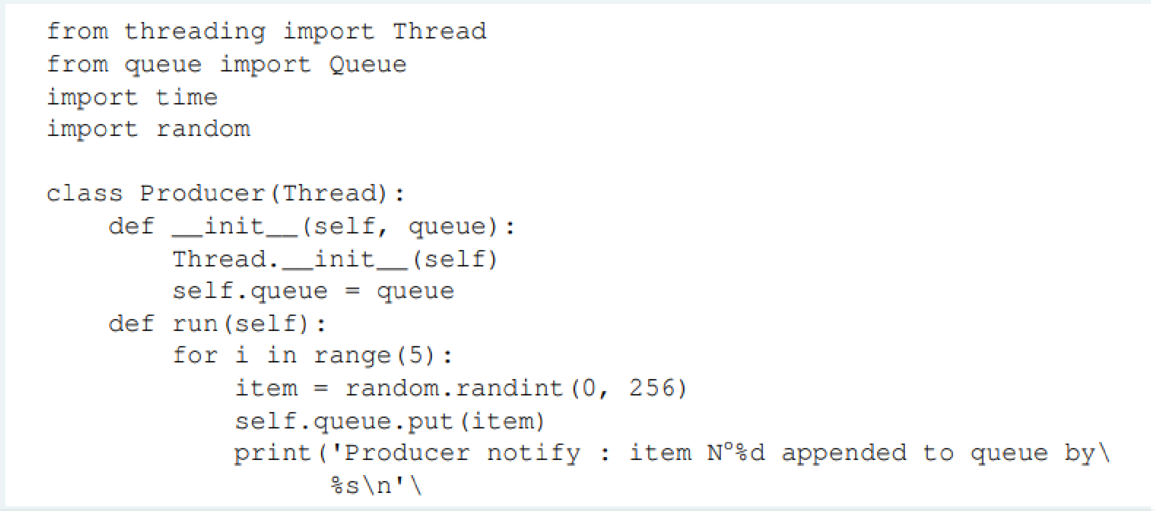
Thread communication using a Queue

- Multithreading can be complicated when threads need to share data or resources. Luckily, the threading module provides many synchronization primitives, including semaphores, condition variables, events, and locks.
- However, it is considered a best practice to use the queue module.
- In fact, a queue is much easier to deal with and makes threaded programming considerably safer, as it effectively funnels all access to a resource of a single thread and allows for a cleaner and more readable design pattern.

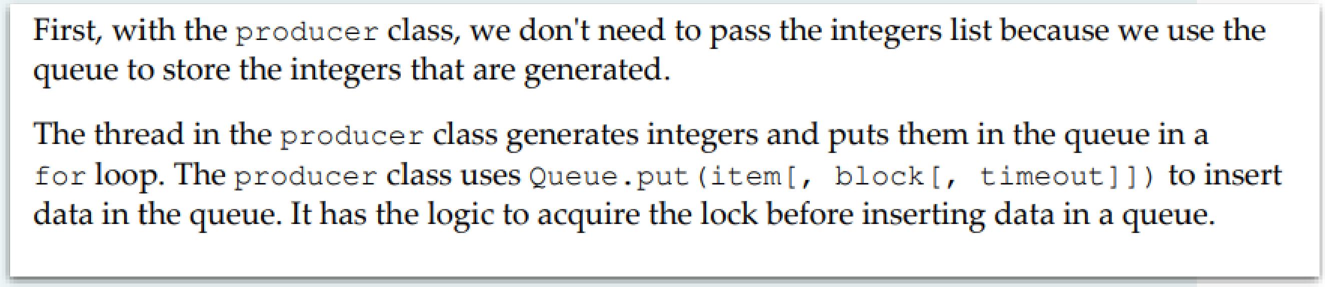

We will simply consider these queue methods:

- `put()`: Puts an item in the queue
- `get()`: Removes and returns an item from the queue
- `task_done()`: Needs to be called each time an item has been processed
- `join()`: Blocks until all items have been processed

Using a Queue (Producer-Consumer)

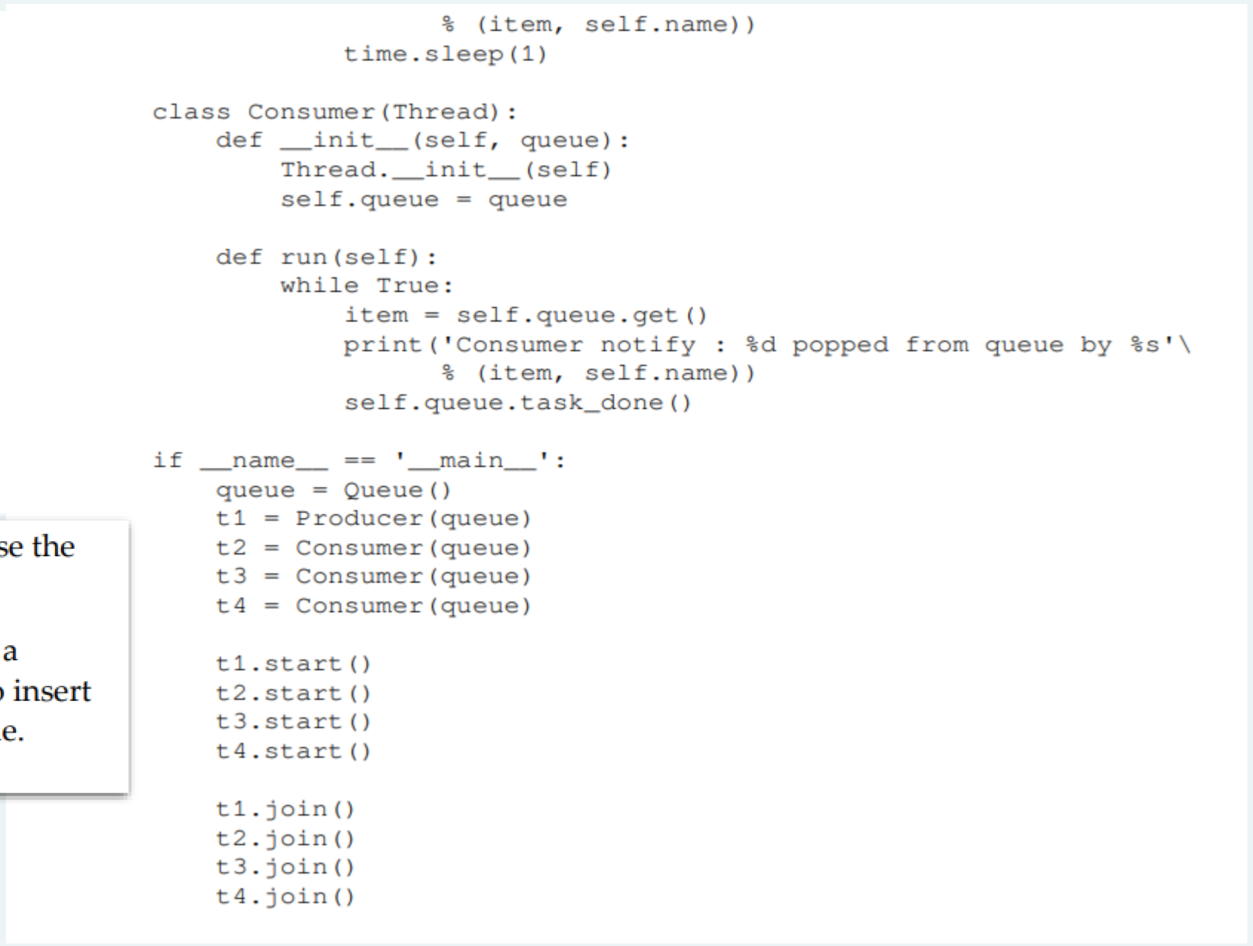


```
from threading import Thread
from queue import Queue
import time
import random

class Producer(Thread):
    def __init__(self, queue):
        Thread.__init__(self)
        self.queue = queue
    def run(self):
        for i in range(5):
            item = random.randint(0, 256)
            self.queue.put(item)
            print('Producer notify : item N°%d appended to queue by\
                  %s\n'\
```



First, with the `producer` class, we don't need to pass the integers list because we use the queue to store the integers that are generated.

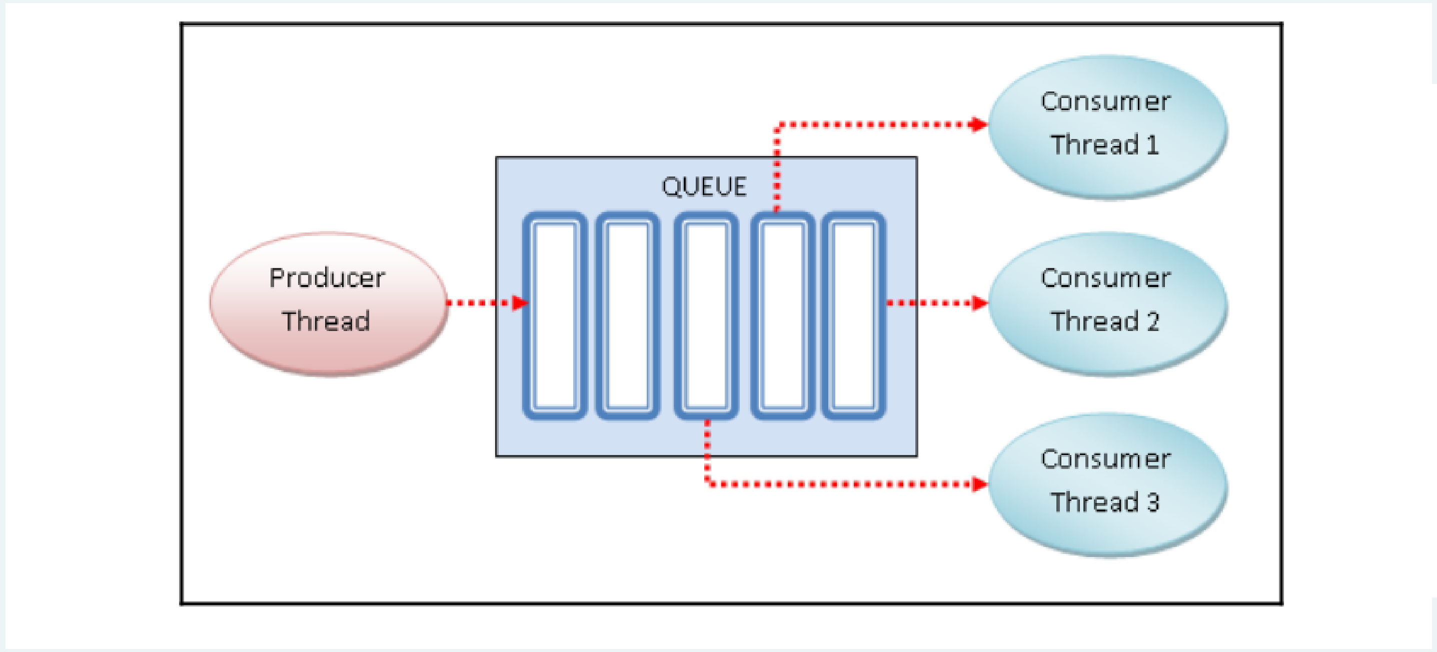
The thread in the `producer` class generates integers and puts them in the queue in a `for` loop. The `producer` class uses `Queue.put(item[, block[, timeout]])` to insert data in the queue. It has the logic to acquire the lock before inserting data in a queue.

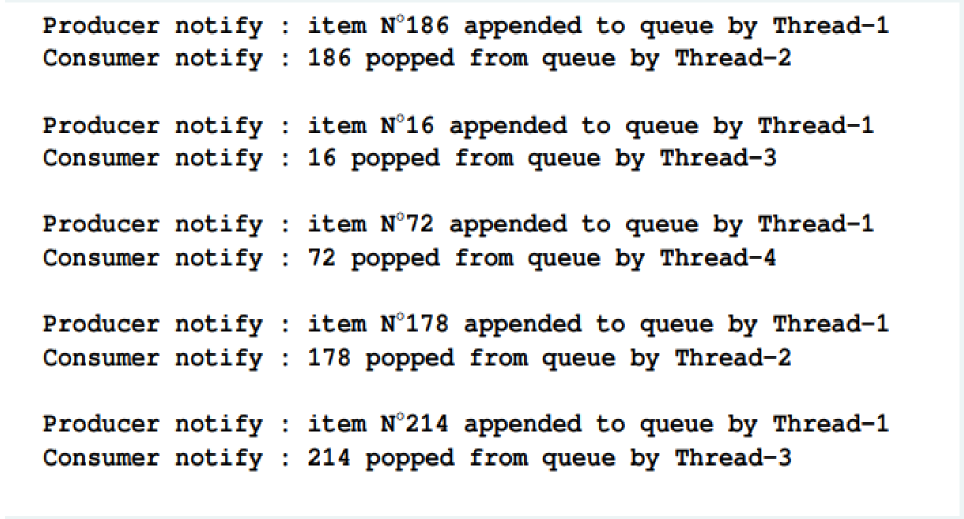


```
                  % (item, self.name))
            time.sleep(1)

class Consumer(Thread):
    def __init__(self, queue):
        Thread.__init__(self)
        self.queue = queue

    def run(self):
        while True:
            item = self.queue.get()
            print('Consumer notify : %d popped from queue by %s'\
                  % (item, self.name))
            self.queue.task_done()

if __name__ == '__main__':
    queue = Queue()
    t1 = Producer(queue)
    t2 = Consumer(queue)
    t3 = Consumer(queue)
    t4 = Consumer(queue)

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

    t1.join()
    t2.join()
    t3.join()
    t4.join()
```

Using a Queue for synchronisation

- The Producer thread acquires the lock and then inserts data in the QUEUE data structure.
- The Consumer threads get the integers from the QUEUE. These threads acquire the lock before removing data from the QUEUE.
- If the QUEUE is empty, then the consumer threads get in a waiting state.





```
Producer notify : item N°186 appended to queue by Thread-1
Consumer notify : 186 popped from queue by Thread-2

Producer notify : item N°16 appended to queue by Thread-1
Consumer notify : 16 popped from queue by Thread-3

Producer notify : item N°72 appended to queue by Thread-1
Consumer notify : 72 popped from queue by Thread-4

Producer notify : item N°178 appended to queue by Thread-1
Consumer notify : 178 popped from queue by Thread-2

Producer notify : item N°214 appended to queue by Thread-1
Consumer notify : 214 popped from queue by Thread-3
```

Multi-processing

- Multiprocessing involves running multiple processes in parallel, each with its own Python interpreter, ideal for CPU-bound tasks that require parallel computation.
- True Parallelism: Each process runs in its own Python interpreter, enabling true parallel computation.
- CPU-bound Tasks: Best suited for tasks that are computationally intensive and benefit from being spread across multiple CPUs or cores.
- This scenario illustrates multiprocessing’s effectiveness in parallel data processing, significantly improving the efficiency of CPU-bound tasks.

Python’s multiprocessing module

- The Python multiprocessing module, which is a part of the standard library of the language, implements the shared memory programming paradigm, that is, the programming of a system that consists of one or more processors that have access to a shared memory.
- Understanding Python's multiprocessing module: Spawning a process, Naming a process, Running processes in the background, Killing a process, Defining a process in a subclass, Using a queue to exchange objects, Using pipes to exchange objects, Synchronizing processes, Managing a state between processes, and Using a process pool.

Multiprocessing: spawning a process

- To create a process, we need to import the multiprocessing module with the following command:

```
import multiprocessing
```

- Each process is associated with the myFunc(i) function. This function outputs the numbers from 0 to i, where i is the ID associated with the process number:

```
def myFunc(i):
 print ('calling myFunc from process n°: %s' %i)
 for j in range (0,i):
 print('output from myFunc is :%s' %j)
```

- Then, we define the process object with myFunc as the target function:

```
if __name__ == '__main__':
 for i in range(6):
 process = multiprocessing.Process(target=myFunc, args=(i,))
```

- Finally, we call the start and join methods on the process created:

```
 process.start()
 process.join()
```

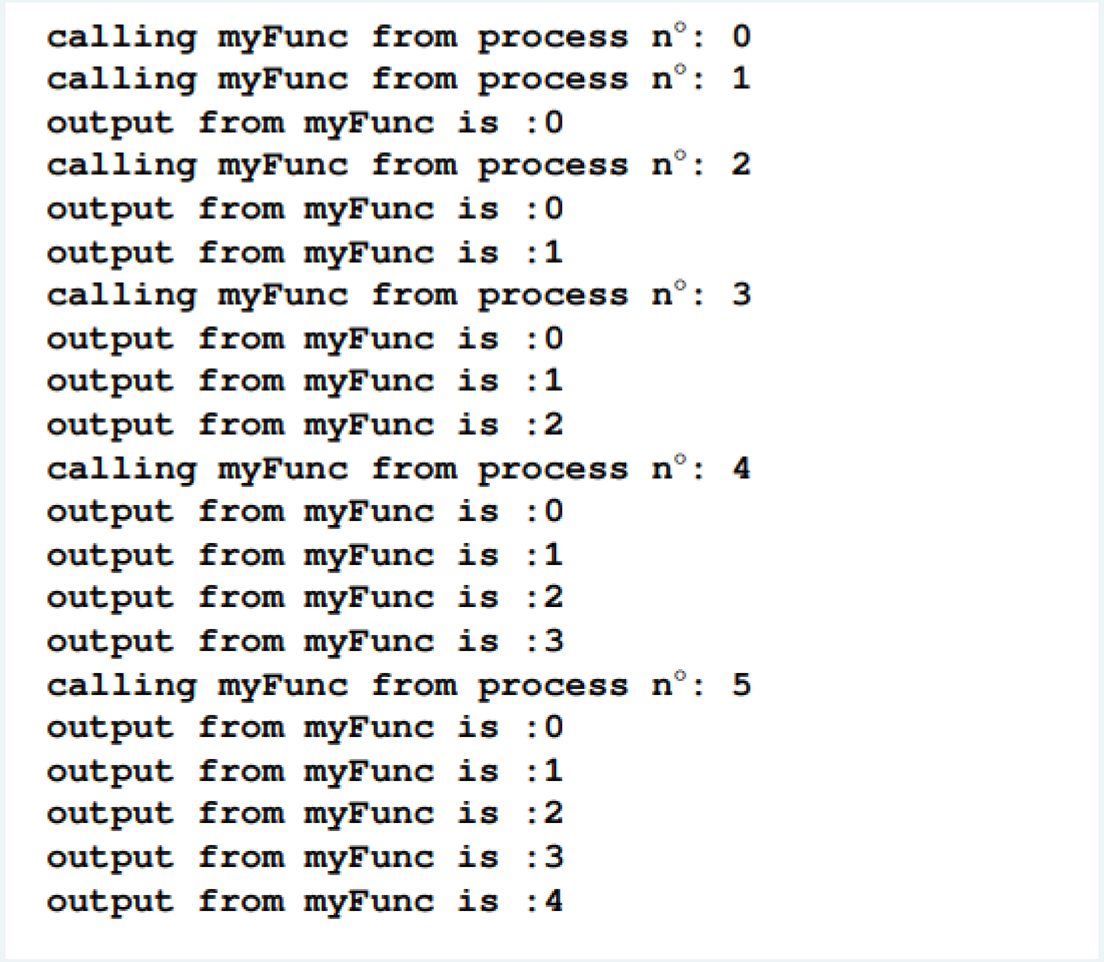
- Without the join method, child processes do not end and must be killed manually.

Spawning a process - Execution

- Python's multiprocessing library allows easy process management by following three simple steps. The first step is the process definition through the multiprocessing class method, Process:

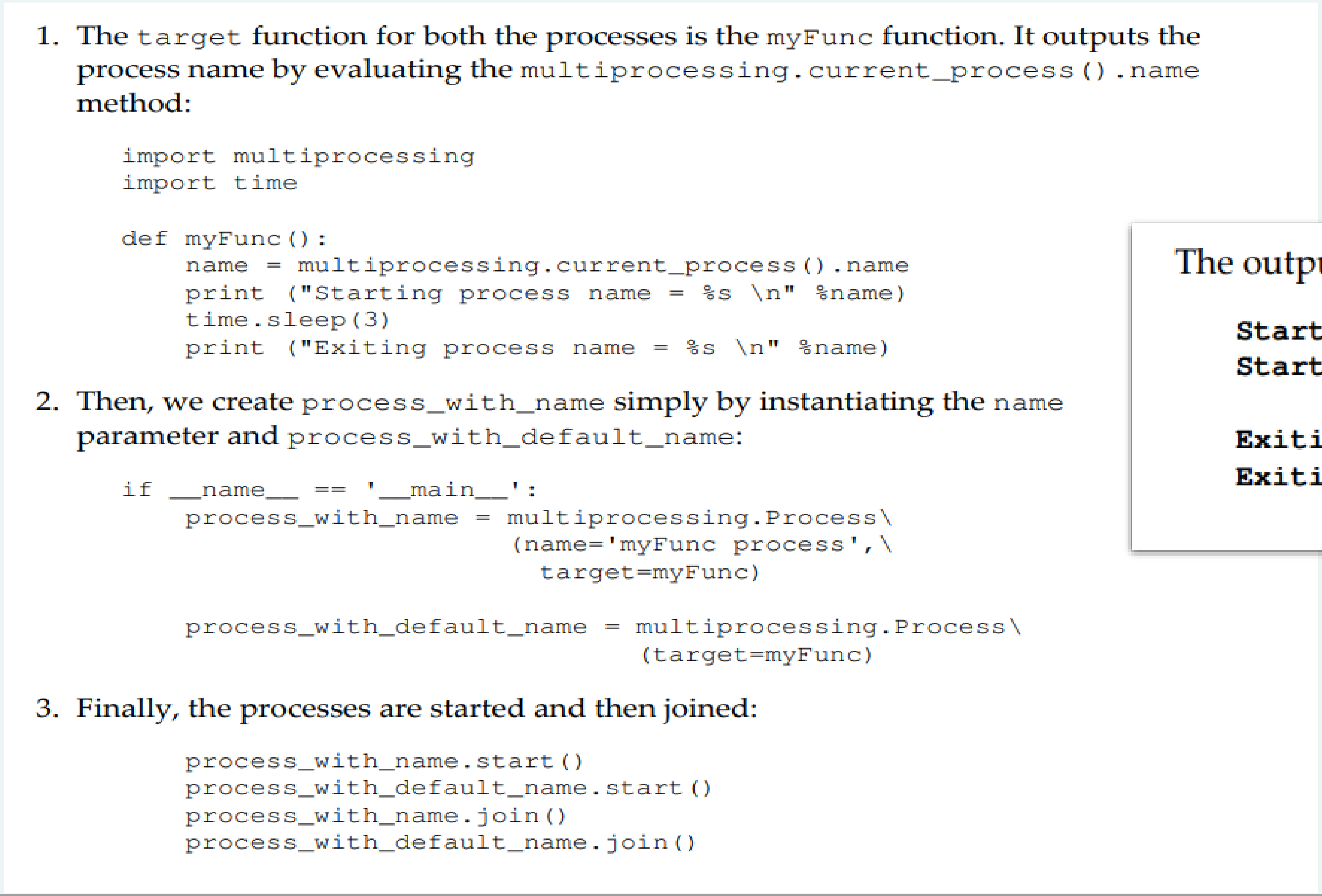
process = multiprocessing.Process(target=myFunc, args=(i,))

- The Process method has as an argument of the function to spawn, *myFunc*, and any arguments of the function itself.



```
calling myFunc from process n°: 0
calling myFunc from process n°: 1
output from myFunc is :0
calling myFunc from process n°: 2
output from myFunc is :0
output from myFunc is :1
calling myFunc from process n°: 3
output from myFunc is :0
output from myFunc is :1
output from myFunc is :2
calling myFunc from process n°: 4
output from myFunc is :0
output from myFunc is :1
output from myFunc is :2
output from myFunc is :3
calling myFunc from process n°: 5
output from myFunc is :0
output from myFunc is :1
output from myFunc is :2
output from myFunc is :3
output from myFunc is :4
```

Naming a process



1. The `target` function for both the processes is the `myFunc` function. It outputs the process name by evaluating the `multiprocessing.current_process().name` method:

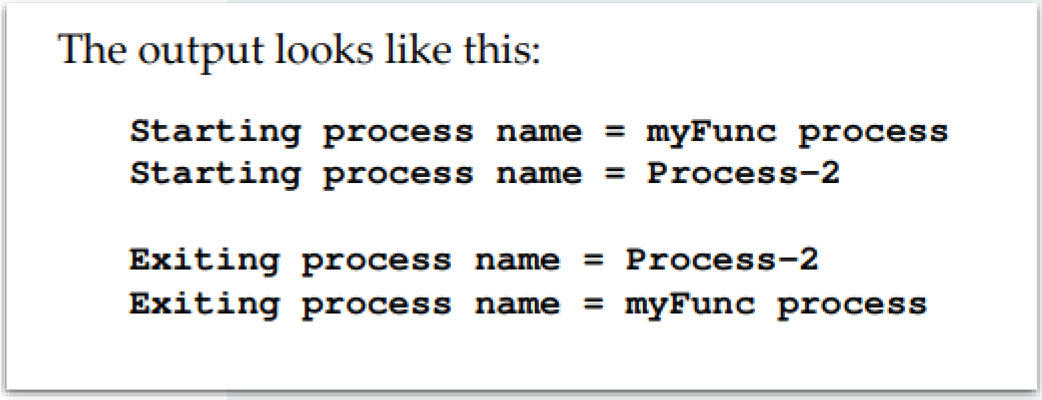
```
import multiprocessing
import time

def myFunc():
    name = multiprocessing.current_process().name
    print ("Starting process name = %s \n" %name)
    time.sleep(3)
    print ("Exiting process name = %s \n" %name)
```

2. Then, we create `process_with_name` simply by instantiating the `name` parameter and `process_with_default_name`:

```
if __name__ == '__main__':
    process_with_name = multiprocessing.Process\
                        (name='myFunc process',\
                         target=myFunc)

    process_with_default_name = multiprocessing.Process\
                                (target=myFunc)
```

3. Finally, the processes are started and then joined:

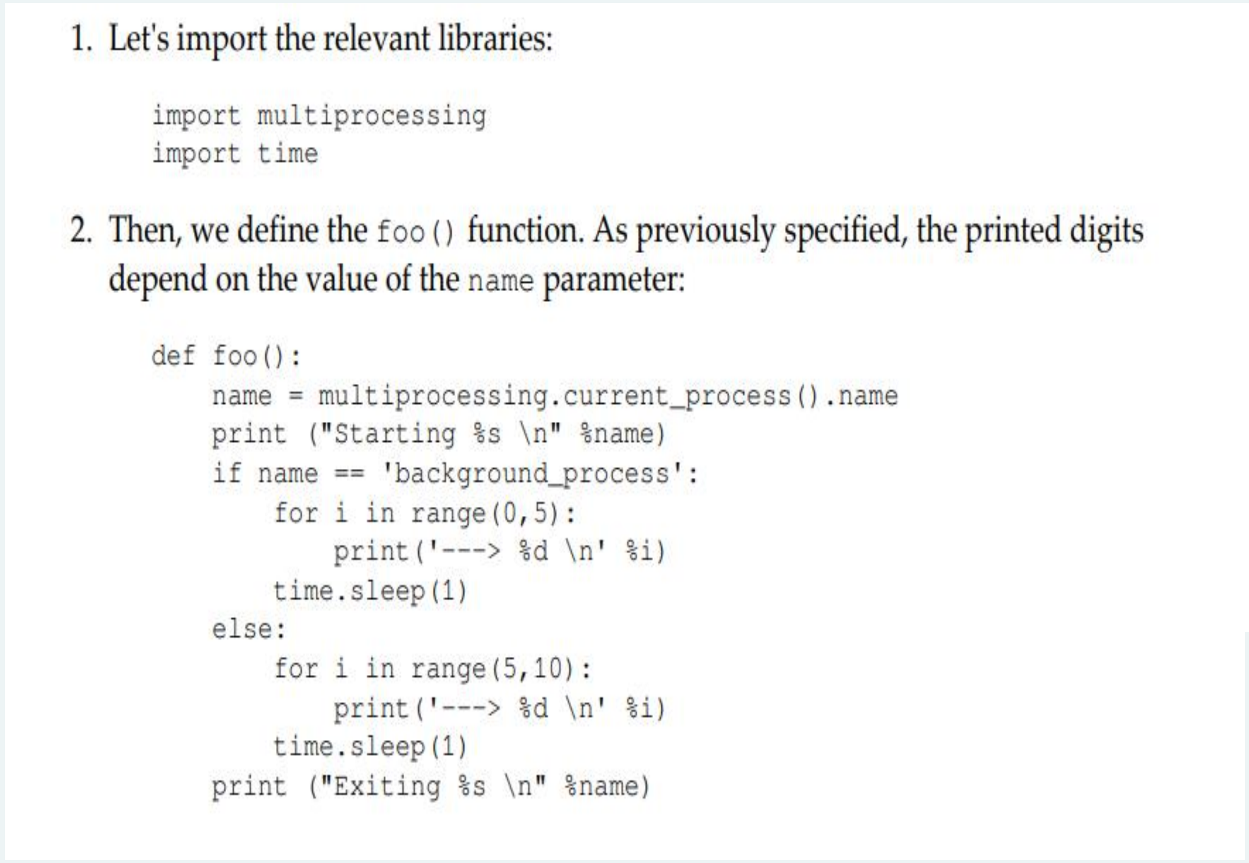
```
    process_with_name.start()
    process_with_default_name.start()
    process_with_name.join()
    process_with_default_name.join()
```



The output looks like this:

```
Starting process name = myFunc process
Starting process name = Process-2

Exiting process name = Process-2
Exiting process name = myFunc process
```

Running a process in the background

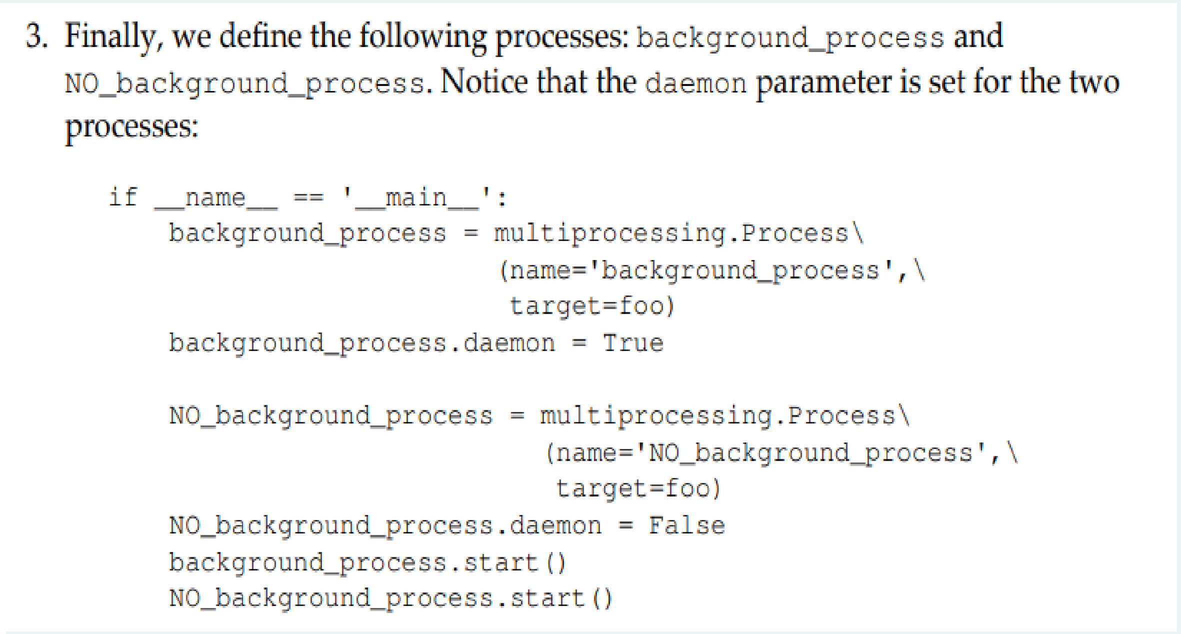


1. Let's import the relevant libraries:

```
import multiprocessing
import time
```

2. Then, we define the `foo()` function. As previously specified, the printed digits depend on the value of the `name` parameter:

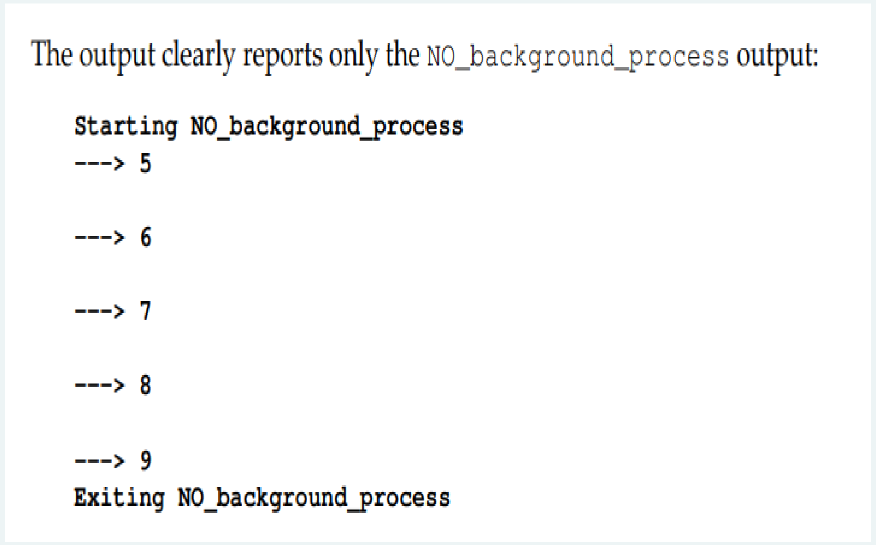
```
def foo():
    name = multiprocessing.current_process().name
    print ("Starting %s \n" %name)
    if name == 'background_process':
        for i in range(0,5):
            print('---> %d \n' %i)
        time.sleep(1)
    else:
        for i in range(5,10):
            print('---> %d \n' %i)
        time.sleep(1)
    print ("Exiting %s \n" %name)
```



3. Finally, we define the following processes: `background_process` and `NO_background_process`. Notice that the `daemon` parameter is set for the two processes:

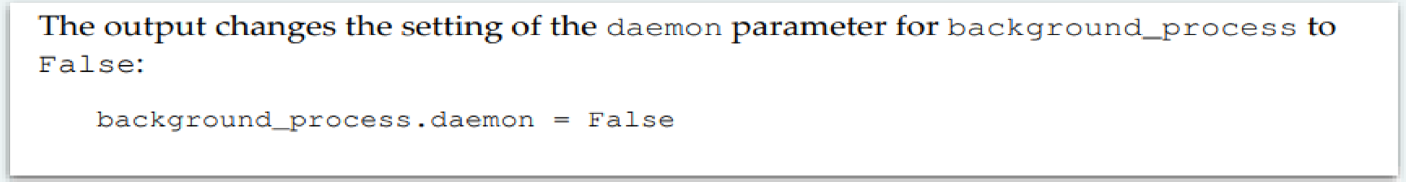
```
if __name__ == '__main__':
    background_process = multiprocessing.Process\
                        (name='background_process',\
                         target=foo)
    background_process.daemon = True

    NO_background_process = multiprocessing.Process\
                           (name='NO_background_process',\
                            target=foo)
    NO_background_process.daemon = False
    background_process.start()
    NO_background_process.start()
```

Running a process in the background



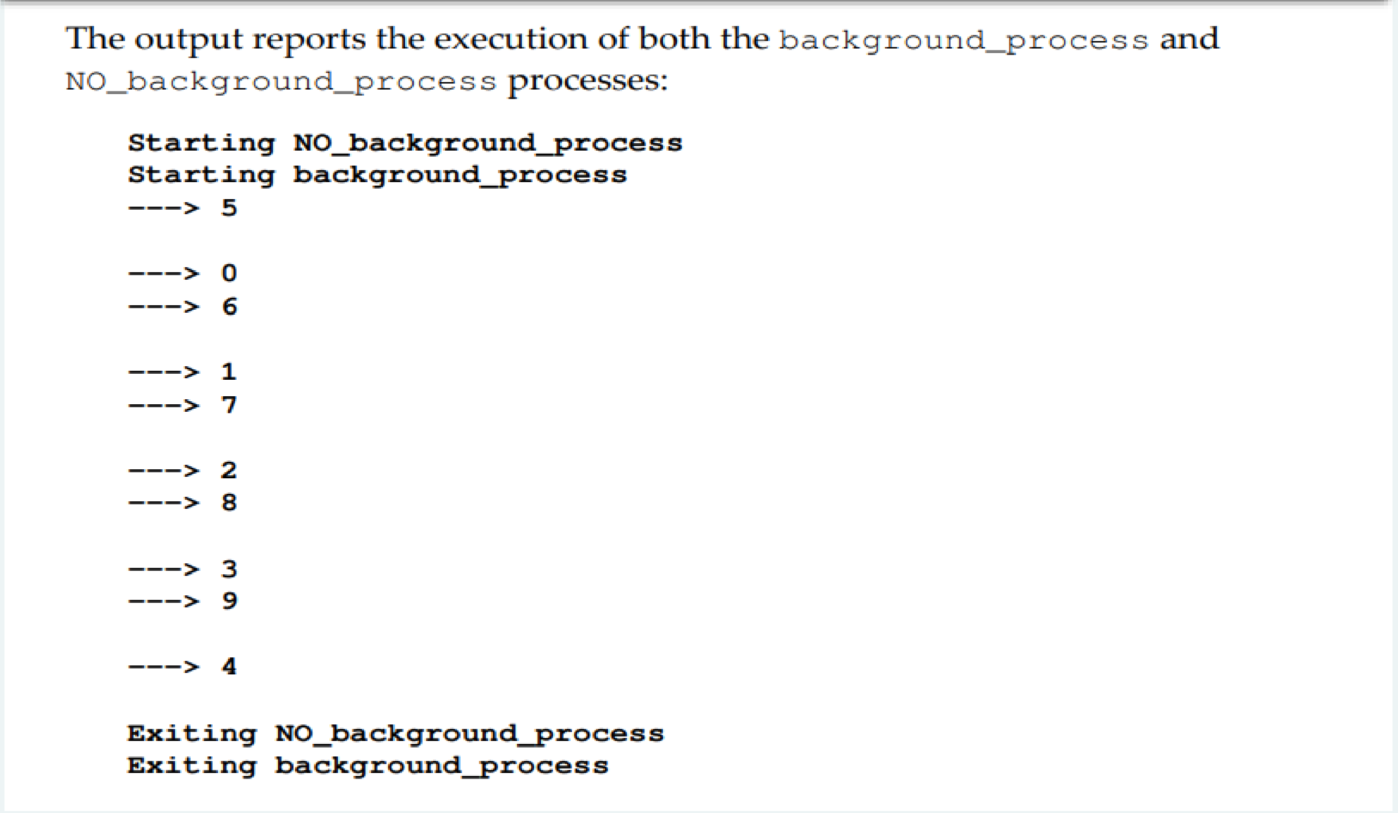
The output clearly reports only the NO_background_process output:

```
    Starting NO_background_process
    ---> 5

    ---> 6

    ---> 7

    ---> 8

    ---> 9
    Exiting NO_background_process
```



The output changes the setting of the daemon parameter for background_process to False:

```
    background_process.daemon = False
```

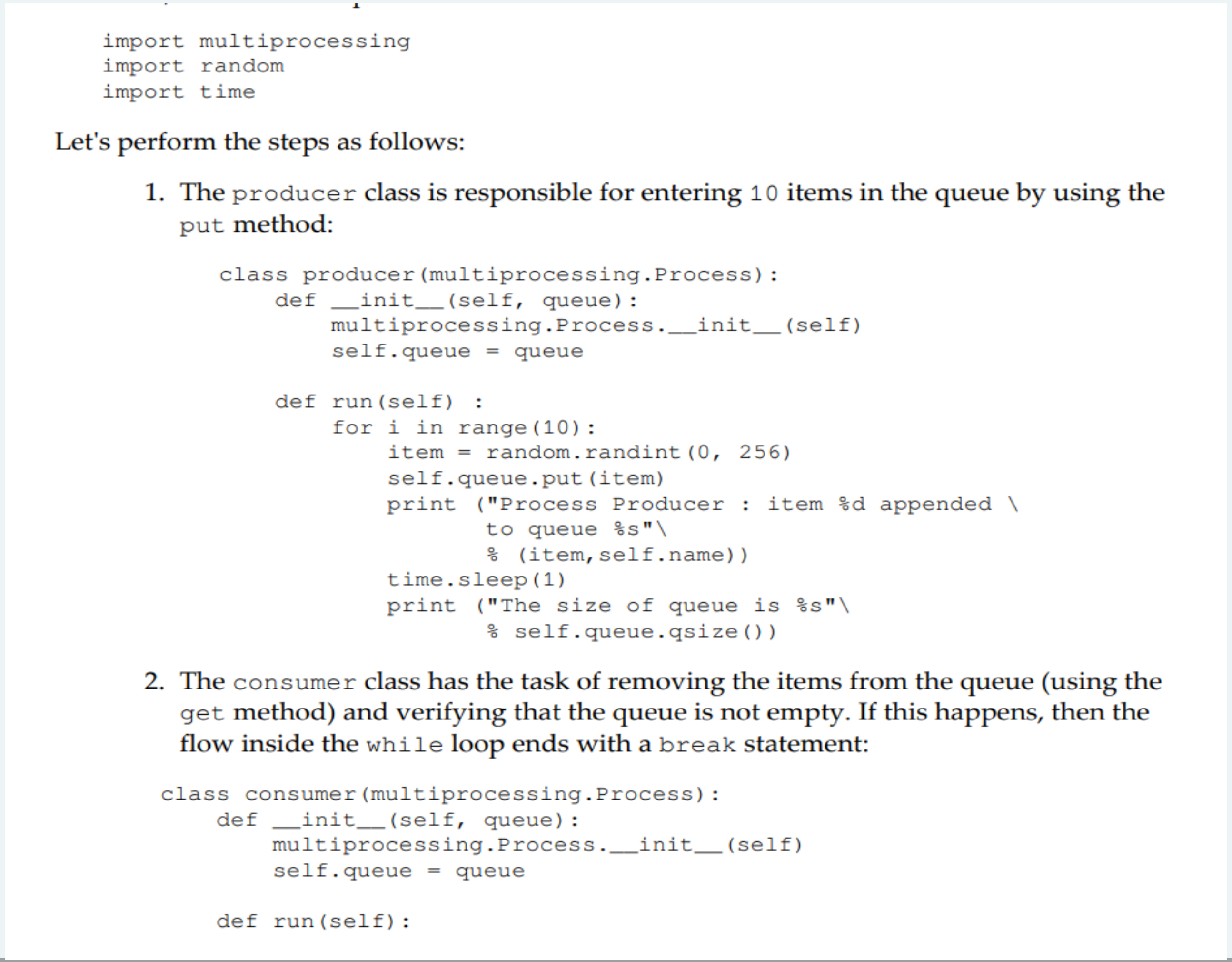


The output reports the execution of both the background_process and NO_background_process processes:

```
    Starting NO_background_process
    Starting background_process
    ---> 5

    ---> 0
    ---> 6

    ---> 1
    ---> 7

    ---> 2
    ---> 8

    ---> 3
    ---> 9

    ---> 4

    Exiting NO_background_process
    Exiting background_process
```

Process synchronisation - Queue

- A queue is a data structure of the First-In, First-Out (FIFO) type (the first input is the first to exit).
- A practical example is the queues to get a service, how to pay at the supermarket, or get your hair cut at the hairdresser.
- Ideally, you are served in the same order as you were presented to. This is exactly how a FIFO queue works.
- **The producer-consumer problem describes two processes: one is the producer, and the other is a consumer, sharing a common buffer of a fixed size.**



```
import multiprocessing
import random
import time
```

Let's perform the steps as follows:

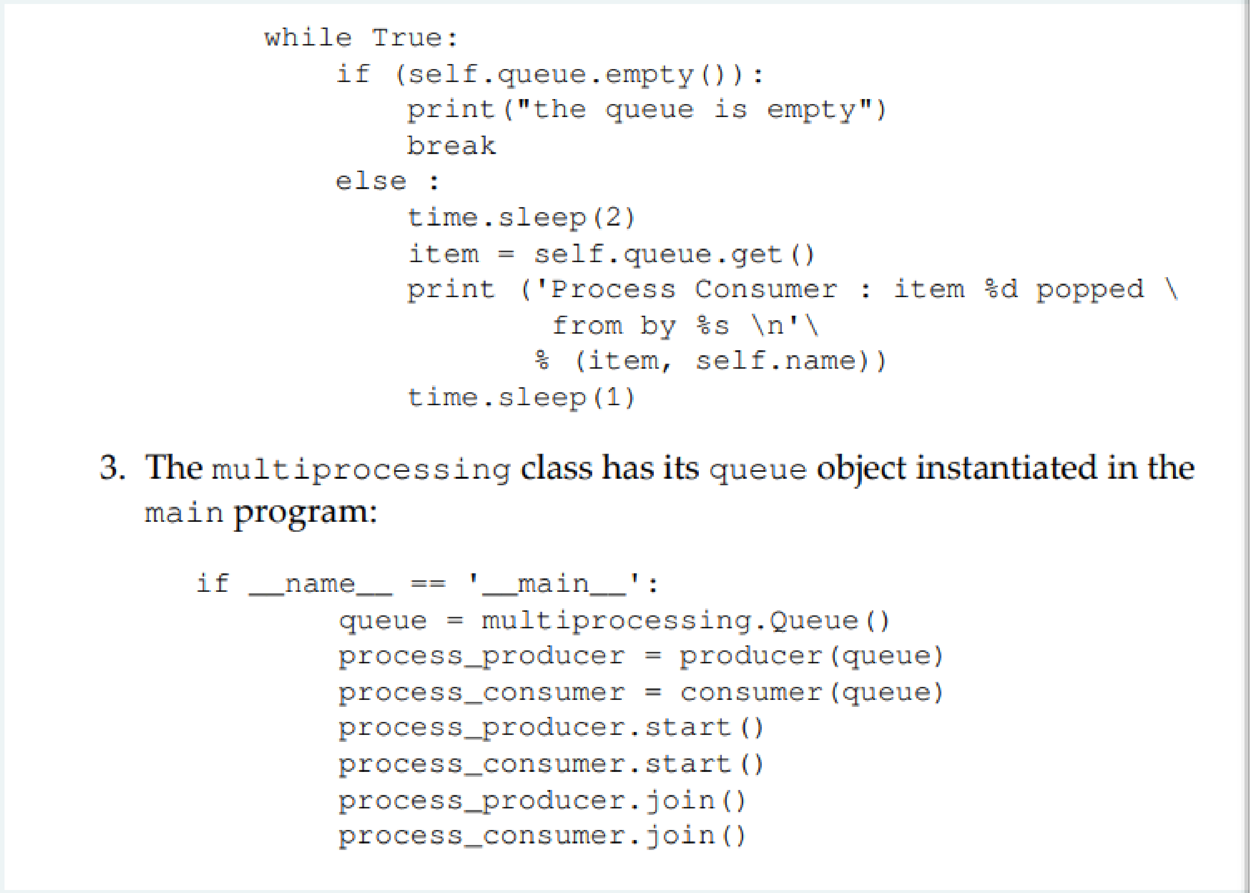
1. The `producer` class is responsible for entering `10` items in the queue by using the `put` method:

```
class producer(multiprocessing.Process):
    def __init__(self, queue):
        multiprocessing.Process.__init__(self)
        self.queue = queue

    def run(self) :
        for i in range(10):
            item = random.randint(0, 256)
            self.queue.put(item)
            print ("Process Producer : item %d appended \
                   to queue %s"\
                   % (item,self.name))
            time.sleep(1)
            print ("The size of queue is %s"\
                   % self.queue.qsize())
```

2. The `consumer` class has the task of removing the items from the queue (using the `get` method) and verifying that the queue is not empty. If this happens, then the flow inside the `while` loop ends with a `break` statement:

```
class consumer(multiprocessing.Process):
    def __init__(self, queue):
        multiprocessing.Process.__init__(self)
        self.queue = queue

    def run(self):
```

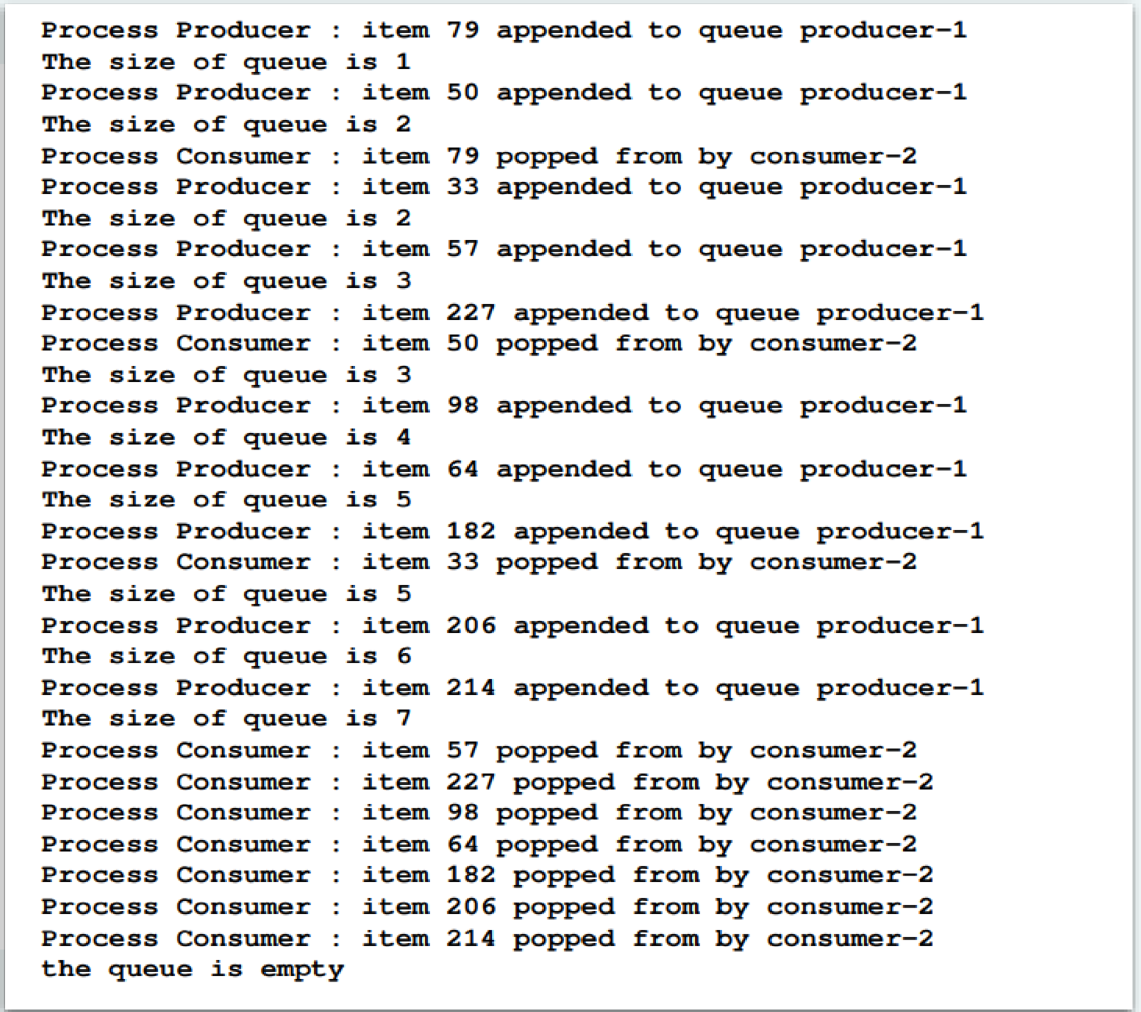
Process synchronisation - Queue



```
            while True:
                if (self.queue.empty()):
                    print("the queue is empty")
                    break
                else :
                    time.sleep(2)
                    item = self.queue.get()
                    print ('Process Consumer : item %d popped \
                            from by %s \n'\
                           % (item, self.name))
                    time.sleep(1)
```

3. The `multiprocessing` class has its `queue` object instantiated in the `main` program:

```
    if __name__ == '__main__':
            queue = multiprocessing.Queue()
            process_producer = producer(queue)
            process_consumer = consumer(queue)
            process_producer.start()
            process_consumer.start()
            process_producer.join()
            process_consumer.join()
```



```
Process Producer : item 79 appended to queue producer-1
The size of queue is 1
Process Producer : item 50 appended to queue producer-1
The size of queue is 2
Process Consumer : item 79 popped from by consumer-2
Process Producer : item 33 appended to queue producer-1
The size of queue is 2
Process Producer : item 57 appended to queue producer-1
The size of queue is 3
Process Producer : item 227 appended to queue producer-1
Process Consumer : item 50 popped from by consumer-2
The size of queue is 3
Process Producer : item 98 appended to queue producer-1
The size of queue is 4
Process Producer : item 64 appended to queue producer-1
The size of queue is 5
Process Producer : item 182 appended to queue producer-1
Process Consumer : item 33 popped from by consumer-2
The size of queue is 5
Process Producer : item 206 appended to queue producer-1
The size of queue is 6
Process Producer : item 214 appended to queue producer-1
The size of queue is 7
Process Consumer : item 57 popped from by consumer-2
Process Consumer : item 227 popped from by consumer-2
Process Consumer : item 98 popped from by consumer-2
Process Consumer : item 64 popped from by consumer-2
Process Consumer : item 182 popped from by consumer-2
Process Consumer : item 206 popped from by consumer-2
Process Consumer : item 214 popped from by consumer-2
the queue is empty
```

Concluding Remarks

- The choice of concurrency model in Python — Asyncio, Threading, or Multiprocessing — depends on the specific problem.
- Asyncio is ideal for I/O-bound tasks, especially involving structured network code.
- Threading can improve the performance of I/O-bound applications where tasks involve blocking I/O operations.
- Multiprocessing is the preferred choice for CPU-bound tasks that require parallel processing.
- Understanding these paradigms' strengths and limitations is key to leveraging them effectively in Python development.

UNIVERSITY OF DERBY

University of Derby, Kedleston Road, Derby, DE22 1GB
T +44 (0)1332 591044 **E** opendays@derby.ac.uk

THANK YOU

derby.ac.uk