

La concurrence en Python

Introduction


Les bases

INTRODUCTION à la concurrence en Python

Application du multithreading
(téléchargement d'images sur le Web)

Gestion de la concurrence :
différents mécanismes de synchronisation

Limites des threads en Python
Problème du GIL (Global Interpreter Lock)



APPLICATION du multithreading

Téléchargement d'images sur le web



Téléchargement multithreadé d'images



tp2_application_sequentialImageDownloading.py

tp2_application_concurrentImageDownloading.py

```
import urllib.request
import time

def downloadImage(imagePath, fileName):
    print("Downloading Image from ", imagePath)
    urllib.request.urlretrieve(imagePath, fileName)
```

séquentiel

```
def main():
    t0 = time.time()

    for i in range(10):
        imageName = "temp/image-" + str(i) + ".jpg"
        downloadImage("https://picsum.photos/640/360", imageName)

    t1 = time.time()
    totalTime = t1 - t0
    print("Total Execution Time {}".format(totalTime))

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

```
import threading
import urllib.request
import time

def downloadImage(imagePath, fileName):
    print("Downloading Image from ", imagePath)
    urllib.request.urlretrieve(imagePath, fileName)
    print("Completed Download")

def executeThread(i):
    imageName = "temp/image-" + str(i) + ".jpg"
    downloadImage("https://picsum.photos/640/360", imageName)

def main():
    t0 = time.time()

    # create an array which will store a reference to
    # all of our threads
    threads = []


    # create 10 threads, append them to our array of threads
    # and start them off
    for i in range(10):
        thread = threading.Thread(target=executeThread, args=(i,))
        threads.append(thread)
        thread.start()

    # ensure that all the threads in our array have completed
    # their execution before we log the total time to complete
    for i in threads:
        i.join()


    # calculate the total execution time
    t1 = time.time()
    totalTime = t1 - t0
    print("Total Execution Time {}".format(totalTime))

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

multithreadé



Gestion de la concurrence : différents mécanismes de synchronisation



Synchronisation : *join()*



tp2_02_threadJoin.py

```
import threading
import time

def ourThread(i):
    print("Thread {} Started".format(i))
    time.sleep(i*2)
    print("Thread {} Finished".format(i))

def main():
    thread = threading.Thread(target=ourThread, args=(1,))
    thread.start()

    print("Is thread 1 Finished?")

    thread2 = threading.Thread(target=ourThread, args=(2,))
    thread2.start()
    thread2.join()

    print("Thread 2 definitely finished")

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

Lock Object - Thread Synchronization in Python

In multithreading when multiple threads are working simultaneously on a shared resource like a file(reading and writing data into a file), then to avoid concurrent modification error(multiple threads accessing same resource leading to inconsistent data) some sort of locking mechanism is used where in when one thread is accessing a resource it takes a lock on that resource and until it releases that lock no other thread can access the same resource.

Lock Object: Python Multithreading

In the `threading` module of Python, for efficient multithreading a primitive lock is used. This lock helps us in the synchronization of two or more threads. Lock class perhaps provides the simplest synchronization primitive in Python.

Primitive lock can have two States: **locked** or **unlocked** and is initially created in unlocked state when we initialize the Lock object. It has two basic methods, `acquire()` and `release()`.

Following is the basic syntax for creating a Lock object:

```
import threading

threading.Lock()
```




tp2_03_lockExample_empty.py

Verrou : *Lock*



tp2_03_lockExample.py

séquentiel

```
import threading
import time
import random

counter = 1

def workerA():
    global counter
    try:
        while counter < 1000:
            counter += 1
            print("Worker A is incrementing counter to {}".format(counter))

    finally:

def workerB():
    global counter
    try:
        while counter > -1000:
            counter -= 1
            print("Worker B is decrementing counter to {}".format(counter))

    finally:

def main():
    t0 = time.time()
    thread1 = threading.Thread(target=workerA)
    thread2 = threading.Thread(target=workerB)

    thread1.start()
    thread2.start()

    thread1.join()
    thread2.join()

    t1 = time.time()

    print("Execution Time {}".format(t1-t0))

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

multithreadé

```
import threading
import time
import random

counter = 1
lock = threading.Lock()

def workerA():
    global counter
    lock.acquire()
    try:
        while counter < 1000:
            counter += 1
            print("Worker A is incrementing counter to {}".format(counter))

    finally:
        lock.release()

def workerB():
    global counter
    lock.acquire()
    try:
        while counter > -1000:
            counter -= 1
            print("Worker B is decrementing counter to {}".format(counter))

    finally:
        lock.release()

def main():
    t0 = time.time()
    thread1 = threading.Thread(target=workerA)
    thread2 = threading.Thread(target=workerB)

    thread1.start()
    thread2.start()

    thread1.join()
    thread2.join()

    t1 = time.time()

    print("Execution Time {}".format(t1-t0))

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


RLock Object: Python Multithreading

An RLock stands for a re-entrant lock. A re-entrant lock can be acquired multiple times by the same thread.

Comme **Lock**, mais on peut appeler plusieurs fois **acquire()** [alors que sinon ça crée une erreur].
Très utile quand on veut mettre des verrous à l'intérieur de fonctions différentes,
elles-mêmes appelées par une fonction (appels en cascade) [voir exemple page suivante].

Verrou : *Rlock* (Reentrant Lock)

tp2_04_locks_empty.py

séquentiel

```
import threading
import time

class myWorker():

    def __init__(self):
        self.a = 1
        self.b = 2

    def modifyA(self):

        print("Modifying A : RLock Acquired: {}".format(self.rlock._is_owned()))
        print("{} ".format(self.rlock))
        self.a = self.a + 1
        time.sleep(5)

    def modifyB(self):

        print("Modifying B : RLock Acquired: {}".format(self.rlock._is_owned()))
        print("{} ".format(self.rlock))
        self.b = self.b - 1
        time.sleep(5)

    def modifyBoth(self):

        print("Rlock acquired, modifying A and B")
        print("{} ".format(self.rlock))
        self.modifyA()
        print("{} ".format(self.rlock))
        self.modifyB()
        print("{} ".format(self.rlock))

workerA = myWorker()
workerA.modifyBoth()
```

tp2_04_locks.py

multithreadé

```
import threading
import time

class myWorker():

    def __init__(self):
        self.a = 1
        self.b = 2
        self.rlock = threading.RLock()

    def modifyA(self):
        with self.rlock:
            print("Modifying A : RLock Acquired: {}".format(self.rlock._is_owned()))
            print("{} ".format(self.rlock))
            self.a = self.a + 1
            time.sleep(5)

    def modifyB(self):
        with self.rlock:
            print("Modifying B : RLock Acquired: {}".format(self.rlock._is_owned()))
            print("{} ".format(self.rlock))
            self.b = self.b - 1
            time.sleep(5)

    def modifyBoth(self):
        with self.rlock:
            print("Rlock acquired, modifying A and B")
            print("{} ".format(self.rlock))
            self.modifyA()
            print("{} ".format(self.rlock))
            self.modifyB()
            print("{} ".format(self.rlock))

workerA = myWorker()
workerA.modifyBoth()
```

Semaphore Objects

This is one of the oldest synchronization primitives in the history of computer science, invented by the early Dutch computer scientist Edsger W. Dijkstra (he used the names `P()` and `V()` instead of `acquire()` and `release()`).

A semaphore manages an internal counter which is decremented by each `acquire()` call and incremented by each `release()` call. The counter can never go below zero; when `acquire()` finds that it is zero, it blocks, waiting until some other thread calls `release()`.

Semaphore Example

Semaphores are often used to guard resources with limited capacity, for example, a database server. In any situation where the size of the resource is fixed, you should use a bounded semaphore. Before spawning any worker threads, your main thread would initialize the semaphore:

```
maxconnections = 5
# ...
pool_sema = BoundedSemaphore(value=maxconnections)
```

Once spawned, worker threads call the semaphore's acquire and release methods when they need to connect to the server:

```
with pool_sema:
    conn = connectdb()
    try:
        # ... use connection ...
    finally:
        conn.close()
```

The use of a bounded semaphore reduces the chance that a programming error which causes the semaphore to be released more than it's acquired will go undetected.

Sémaphore : *Semaphore*

tp2_06_semaphores_empty.py

séquentiel

```
import threading
import time
import random

class TicketSeller(threading.Thread):
    ticketsSold = 0

    def __init__(self):
        threading.Thread.__init__(self);

        print("Ticket Seller Started Work")

    def run(self):
        global ticketsAvailable
        running = True
        while running:
            self.randomDelay()

            if(ticketsAvailable <= 0):
                running = False
            else:
                self.ticketsSold = self.ticketsSold + 1
                ticketsAvailable = ticketsAvailable - 1
                print("{} Sold One ({} left)".format(self.getName(), ticketsAvailable))

        print("Ticket Seller {} Sold {} tickets in total".format(self.getName(), self.ticketsSold))

    def randomDelay(self):
        time.sleep(random.randint(0,4)/4)

# our semaphore primitive

# Our Ticket Allocation
ticketsAvailable = 200

# our array of sellers
sellers = []
for i in range(4):
    seller = TicketSeller()
    seller.start()
    sellers.append(seller)

# joining all our sellers
for seller in sellers:
    seller.join()
```

tp2_06_semaphores.py

multithreadé

```
import threading
import time
import random

class TicketSeller(threading.Thread):
    ticketsSold = 0

    def __init__(self, semaphore):
        threading.Thread.__init__(self);
        self.sem = semaphore
        print("Ticket Seller Started Work")

    def run(self):
        global ticketsAvailable
        running = True
        while running:
            self.randomDelay()

            self.sem.acquire()
            if(ticketsAvailable <= 0):
                running = False
            else:
                self.ticketsSold = self.ticketsSold + 1
                ticketsAvailable = ticketsAvailable - 1
                print("{} Sold One ({} left)".format(self.getName(), ticketsAvailable))
            self.sem.release()

        print("Ticket Seller {} Sold {} tickets in total".format(self.getName(), self.ticketsSold))

    def randomDelay(self):
        time.sleep(random.randint(0,4)/4)

# our semaphore primitive
semaphore = threading.Semaphore()

# Our Ticket Allocation
ticketsAvailable = 200

# our array of sellers
sellers = []
for i in range(4):
    seller = TicketSeller(semaphore)
    seller.start()
    sellers.append(seller)

# joining all our sellers
for seller in sellers:
    seller.join()
```



tp2_06_semaphores.py

Sémaphore : *BoundedSemaphore*



tp2_07_boundedSemaphores.py

multithreadé

```
import threading
import time
import random

class TicketSeller(threading.Thread):
    ticketsSold = 0

    def __init__(self, semaphore):
        threading.Thread.__init__(self);
        self.sem = semaphore
        print("Ticket Seller Started Work")

    def run(self):
        global ticketsAvailable
        running = True
        while running:
            self.randomDelay()

            self.sem.acquire()
            if(ticketsAvailable <= 0):
                running = False
            else:
                self.ticketsSold = self.ticketsSold + 1
                ticketsAvailable = ticketsAvailable - 1
                print("{} Sold One ({} left)".format(self.getName(), ticketsAvailable))
            self.sem.release()
            print("Ticket Seller {} Sold {} tickets in total".format(self.getName(), self.ticketsSold))

    def randomDelay(self):
        time.sleep(random.randint(0,4)/4)

# our semaphore primitive
semaphore = threading.Semaphore()
# Our Ticket Allocation
ticketsAvailable = 200

# our array of sellers
sellers = []
for i in range(4):
    seller = TicketSeller(semaphore)
    seller.start()
    sellers.append(seller)

# joining all our sellers
for seller in sellers:
    seller.join()
```

multithreadé

```
import threading
import time
import random

class TicketSeller(threading.Thread):
    ticketsSold = 0

    def __init__(self, semaphore):
        threading.Thread.__init__(self);
        self.sem = semaphore
        print("Ticket Seller Started Work")

    def run(self):
        global ticketsAvailable
        running = True
        while running:
            self.randomDelay()

            self.sem.acquire()
            if(ticketsAvailable <= 0):
                running = False
            else:
                self.ticketsSold = self.ticketsSold + 1
                ticketsAvailable = ticketsAvailable - 1
                print("{} Sold One ({} left)".format(self.getName(), ticketsAvailable))
            self.sem.release()
            print("Ticket Seller {} Sold {} tickets in total".format(self.getName(), self.ticketsSold))

    def randomDelay(self):
        time.sleep(random.randint(0,4)/4)

# our semaphore primitive
semaphore = threading.BoundedSemaphore(2)
# Our Ticket Allocation
ticketsAvailable = 200

# our array of sellers
sellers = []
for i in range(4):
    seller = TicketSeller(semaphore)
    seller.start()
    sellers.append(seller)

# joining all our sellers
for seller in sellers:
    seller.join()
```

Condition Object - Thread Synchronization in Python

In order to synchronize the access to any resources more efficiently, we can associate a condition with tasks, for any thread to wait until a certain condition is met or notify other threads about the condition being fulfilled so that they may unblock themselves.

Let's take a simple example to understand this. In the **Producer Consumer problem**, if there is one Producer producing some item and one Consumer consuming it, then until the Producer has produced the item the Consumer cannot consume it. Hence the Consumer waits until the Producer produces an item. And it's the duty of the Producer to inform the Consumer that an item is available for consumption once it is successfully produced.

And if there are multiple Consumers consuming the item produced by the Producer then the Producer must inform all the Consumers about the new item produced.

This is a perfect usecase for the condition object in multithreading in python.

Condition : *Condition*

Example :

Producteur / Consommateur

séquentiel

```
class Publisher(threading.Thread):  
  
    def __init__(self, integers):  
  
        self.integers = integers  
        threading.Thread.__init__(self)  
  
    def run(self):  
        while True:  
            integer = random.randint(0,1000)  
  
            print("Condition Acquired by Publisher: {}".format(self.name))  
            self.integers.append(integer)  
            print("Publisher {} appending to array: {}".format(self.name, integer))  
  
            print("Condition Released by Publisher: {}".format(self.name))  
  
            time.sleep(1)  
  
class Subscriber(threading.Thread):  
  
    def __init__(self, integers):  
        self.integers = integers  
  
        threading.Thread.__init__(self)  
  
    def run(self):  
        while True:  
  
            print("Condition Acquired by Consumer: {}".format(self.name))  
            while True:  
                if self.integers:  
                    integer = self.integers.pop()  
                    print("{} Popped from list by Consumer: {}".format(integer, self.name))  
                    break  
                print("Condition Wait by {}".format(self.name))  
  
            print("Consumer {} Releasing Condition".format(self.name))  
  
def main():  
    integers = []  
  
    # Our Publisher  
    publ = Publisher(integers)  
    publ.start()  
  
    # Our Subscribers  
    sub1 = Subscriber(integers)  
    sub2 = Subscriber(integers)  
    sub1.start()  
    sub2.start()  
  
    ## Joining our Threads  
    publ.join()  
    consumer1.join()  
    consumer2.join()
```

tp2_09_pubSub_empty.py

multithreadé

```
class Publisher(threading.Thread):  
  
    def __init__(self, integers, condition):  
        self.condition = condition  
        self.integers = integers  
        threading.Thread.__init__(self)  
  
    def run(self):  
        while True:  
            integer = random.randint(0,1000)  
            self.condition.acquire()  
            print("Condition Acquired by Publisher: {}".format(self.name))  
            self.integers.append(integer)  
            print("Publisher {} appending to array: {}".format(self.name, integer))  
            self.condition.notify()  
            print("Condition Released by Publisher: {}".format(self.name))  
            self.condition.release()  
            time.sleep(1)  
  
class Subscriber(threading.Thread):  
  
    def __init__(self, integers, condition):  
        self.integers = integers  
        self.condition = condition  
        threading.Thread.__init__(self)  
  
    def run(self):  
        while True:  
            self.condition.acquire()  
            print("Condition Acquired by Consumer: {}".format(self.name))  
            while True:  
                if self.integers:  
                    integer = self.integers.pop()  
                    print("{} Popped from list by Consumer: {}".format(integer, self.name))  
                    break  
                print("Condition Wait by {}".format(self.name))  
                self.condition.wait()  
            print("Consumer {} Releasing Condition".format(self.name))  
            self.condition.release()  
  
def main():  
    integers = []  
    condition = threading.Condition()  
  
    # Our Publisher  
    publ = Publisher(integers, condition)  
    publ.start()  
  
    # Our Subscribers  
    sub1 = Subscriber(integers, condition)  
    sub2 = Subscriber(integers, condition)  
    sub1.start()  
    sub2.start()  
  
    ## Joining our Threads  
    publ.join()  
    consumer1.join()  
    consumer2.join()
```

tp2_09_pubSub.py

Python Multithreading: Event Object

The Event class object provides a simple mechanism which is used for communication between threads where one thread signals an event while the other threads wait for it. So, when one thread which is intended to produce the signal produces it, then the waiting thread gets activated.

An internal flag is used by the event object known as the **event flag** which can be set as true using the `set()` method and it can be reset to false using the `clear()` method.

The `wait()` method blocks a thread until the event flag for which it is waiting is set true by any other thread..



tp2_10_events_empty.py

Événement : *Event*



tp2_10_events.py

séquentiel

```
import threading
import time

def myThread():
    print("Waiting for Event to be set")
    time.sleep(1)
    print("myEvent has been set")

def main():
    thread1 = threading.Thread(target=myThread)
    thread1.start()

    time.sleep(10)

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

multithreadé

```
import threading
import time

def myThread(myEvent):
    while not myEvent.is_set():
        print("Waiting for Event to be set")
        time.sleep(1)
    print("myEvent has been set")

def main():
    myEvent = threading.Event()
    thread1 = threading.Thread(target=myThread, args=(myEvent,))
    thread1.start()

    time.sleep(10)
    myEvent.set()

if __name__ == '__main__':
    main|()
```

Barrier Object - Python Multithreading

Barrier object is created by using `Barrier` class which is available in the `threading` module. This object can be used where we want a set of threads to wait for each other.

For example, if we have two threads and we want both the threads to execute when both are ready. In such situation both the threads will call the `wait()` method on the barrier object once they are ready and both the threads will be released simultaneously only when both of them have called the `wait()` method.



tp2_11_barriers_empty.py

Barrière : *Barrier*

Barrier : bloque tous les threads qui appellent **wait()**.
Permet de synchroniser des threads à des endroits précis,
à l'intérieur de la méthode **run()**.
join() attend la fin de la méthode **run()**.



tp2_11_barriers.py

séquentiel

```
import threading
import time
import random

class myThread(threading.Thread):

    def __init__(self):
        threading.Thread.__init__(self)

    def run(self):
        print("Thread {} working on something".format(threading.current_thread()))
        time.sleep(random.randint(1,10))
        print("Thread {} is joining {} waiting on Barrier".format(
            threading.current_thread()))

        print("Barrier has been lifted, continuing with work")

threads = []

for i in range(4):
    thread = myThread()
    thread.start()
    threads.append(thread)

for t in threads:
    t.join()
```

multithreadé

```
import threading
import time
import random

class myThread(threading.Thread):

    def __init__(self, barrier):
        threading.Thread.__init__(self)
        self.barrier = barrier;

    def run(self):
        print("Thread {} working on something".format(threading.current_thread()))
        time.sleep(random.randint(1,10))
        print("Thread {} is joining {} waiting on Barrier".format(
            threading.current_thread(), self.barrier.n_waiting))

        self.barrier.wait()

        print("Barrier has been lifted, continuing with work")

barrier = threading.Barrier(4)

threads = []

for i in range(4):
    thread = myThread(barrier)
    thread.start()
    threads.append(thread)

for t in threads:
    t.join()
```

n_waiting: number of threads
currently waiting in the barrier



Limites des threads en Python

Problème du GIL (Global Interpreter Lock)



Problème du GIL en Python

The Python Global Interpreter Lock or **GIL**, in simple words, is a mutex (or a lock) that allows only one **thread** to hold the control of the Python interpreter.

This means that only one thread can be in a state of execution at any point in time. The impact of the GIL isn't visible to developers who execute single-threaded programs, but it can be a performance bottleneck in CPU-bound and multi-threaded code.

Since the GIL allows only one thread to execute at a time even in a multi-threaded architecture with more than one CPU core, the GIL has gained a reputation as an “infamous” feature of Python.

Problème du GIL en Python

Impact sur les programmes multithreadés en Python

When you look at a typical Python program—or any computer program for that matter—there's a difference between those that are CPU-bound in their performance and those that are I/O-bound.

CPU-bound programs are those that are pushing the CPU to its limit. This includes programs that do mathematical computations like matrix multiplications, searching, image processing, etc.

I/O-bound programs are the ones that spend time waiting for **Input/Output** which can come from a user, file, database, network, etc. I/O-bound programs sometimes have to wait for a significant amount of time till they get what they need from the source due to the fact that the source may need to do its own processing before the input/output is ready, for example, a user thinking about what to enter into an input prompt or a database query running in its own process.

Problème du GIL en Python

How to Deal With Python's GIL

If the GIL is causing you problems, here a few approaches you can try:

Multi-processing vs multi-threading: The most popular way is to use a multi-processing approach where you use multiple processes instead of threads. Each Python process gets its own Python interpreter and memory space so the GIL won't be a problem. Python has a `multiprocessing` module which lets us create processes easily like this:

Le module ***multiprocessing*** permet au développeur d'exploiter au mieux tous les processeurs d'une machine.

Multi-Threading en Python

Good for **I/O bounded** applications
Ex : downloading images (web request),
file access, ...

```
import threading
import time

def myChildThread():
    print("Child Thread Starting")
    time.sleep(5)

child = threading.Thread(target=myChildThread)
child.start()
```



Multi-Processing en Python

Good for **CPU bounded** applications
Ex : mathematical computations (matrix, image processing,...)

```
from multiprocessing import Process
import time

def myWorker():
    t1 = time.time()
    print("Process started at: {}".format(t1))
    time.sleep(20)

myProcess = Process(target=myWorker)
print("Process {}".format(myProcess))
myProcess.start()
```

API très proche

La concurrence en Python

On peut faire bien plus avec les threads et la concurrence en Python.

On a juste vu les notions de base.