

(/dashboard)

 RUN CELL RUN ALL

(834)

(831)

(/hub/dashboard)

 VALIDATE SAVERobin  
BIRON

()

DataScientest • com

(/hub/cheatsheet)



(/hub/progress)



(/typo)



(/forum/271)

# Python avancé

## Multithreading et multiprocessing

### 1. Définitions



Pour optimiser le temps d'exécution d'un programme il est souvent pertinent d'avoir recours à la programmation concurrente ou parallèle plutôt que

(/dashboard)



RUN CELL



RUN ALL



SAVE



Éléments essentiels



( )

(834)

(831)

(/hub/dashboard)



VALIDATE

(/hub/cheatsheet)



(/hub/progress)



(/typo)



(/forum/271)

Un **processeur** (ou CPU pour Central Processing Unit), est l'un des composants électroniques essentiels que l'on trouve dans nos ordinateurs et qui se charge de l'**exécution d'instructions** qui lui sont données. Un processeur est principalement défini par 2 caractéristiques : Sa **fréquence** et son **nombre de cœurs**. La fréquence associée au processeur correspond au **nombre de cycles** qu'il est capable de réaliser par seconde. Un processeur peut avoir **un ou plusieurs cœurs** qui correspondent à des **unités de calcul**. Un processeur **multicœur** pourra exécuter plusieurs tâches simultanément, si ces tâches le permettent, en répartissant les tâches par cœur disponible.

La **RAM** pour Random Access Memory correspond à la mémoire vive d'un ordinateur. C'est un espace de stockage temporaire. Le système accède à cette mémoire de façon instantanée ce qui permet la fluidité de l'interface.

### Processus et thread

On appelle **processus** un programme autrement dit un ensemble d'instructions qui est **en cours d'exécution**. L'ensemble des instructions d'un programme est stocké sur la **RAM** de telle sorte à ce que le processeur puisse comprendre ces instructions. A partir de la RAM, le processeur va donc exécuter chaque instruction. On parle de cycle "*fetch-execute*", cela signifie que le processeur cherche l'instruction puis l'exécute. Il est important de noter que chaque processus est indépendant des autres existants.

Un **thread** est une sous-partie de processus et correspond à un **fil d'exécution** d'instructions. Ainsi un processus pourra être composé d'un ou plusieurs threads, qui seront exécutés de façon parallèle ou concurrente.

Processus et threads sont ainsi des notions très proches mais elles diffèrent sur des éléments clés. Chaque **processus** dispose d'un **espace mémoire propre** qui lui est alloué. A contrario, tous les threads appartenant à un même processus **partagent un espace mémoire**. Ces propriétés concernant l'allocation de la mémoire influent notamment sur la communication et la rapidité de passage d'un processus ou thread à un autre. En effet, la communication entre **threads** sera par exemple **moins coûteuse** qu'une communication entre processus et il sera également plus rapide de passer d'un thread à un autre plutôt qu'un processus à un autre. Il est également à noter que la création d'un thread sera moins coûteuse en temps que la création d'un processus à cause de l'allocation de cet espace mémoire propre.

### Exécution synchrone et asynchrone

On parle d'exécution **synchrone** lorsqu'un programme est exécuté en

(/dashboard)



RUN

**séquentiel.** Si l'on considère un fil d'instructions, alors l'exécution de chaque instruction est conditionnée à la fin de l'exécution de l'instruction précédente. Il est important de noter que les instructions portent ici un ordre. L'illustration ci-dessous illustre le fonctionnement en séquentiel.

(834)

(831)

(/hub/dashboard)



VALIDATE



SAVE

### Exécution séquentielle

Robin  
BIRON

(/hub/cheatsheet)



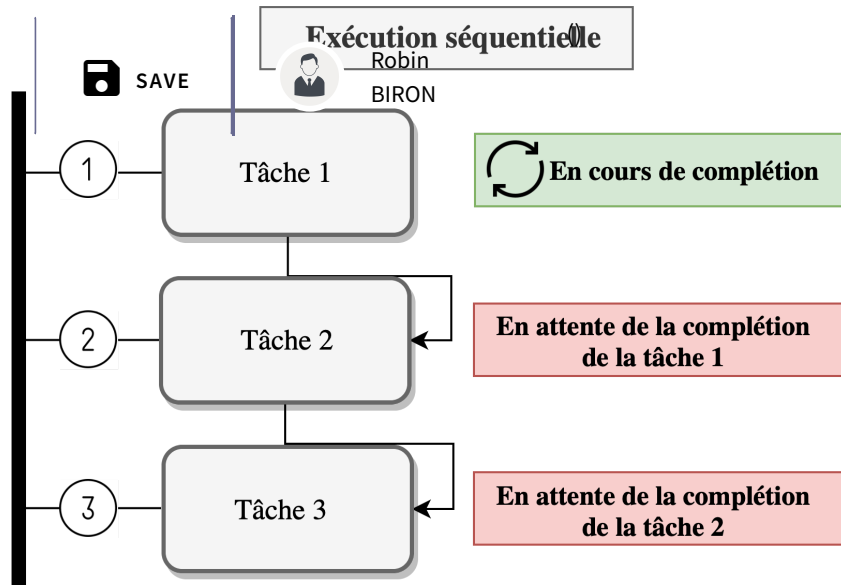
(/hub/progress)



(/typo)



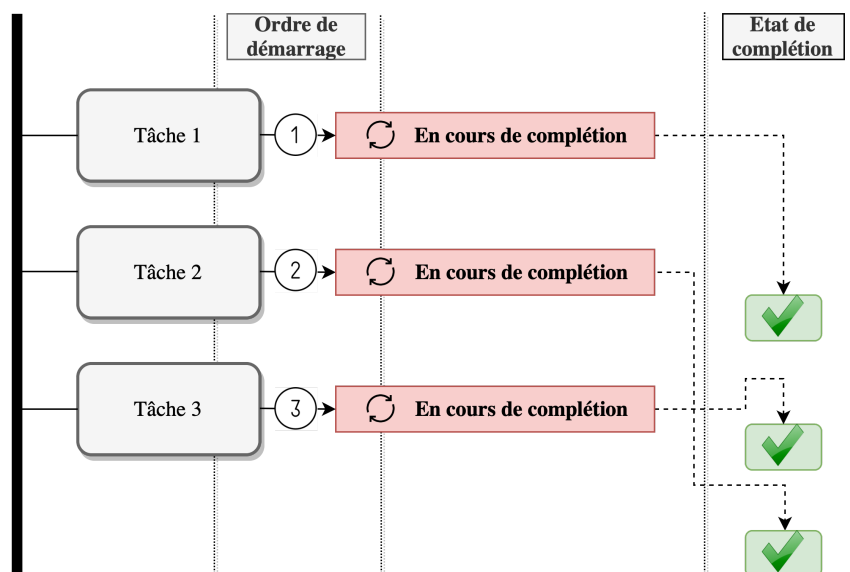
(/forum/271)



Dans cet exemple, le processus 2 attend la complétion du processus 1 pour démarrer. De même le processus 3 attend la complétion du processus 2 pour démarrer.

L'exécution **asynchrone** correspond au fait d'exécuter des nouvelles instructions alors que les instructions précédentes n'ont pas encore fini leur exécution : l'exécution d'une nouvelle instruction ne sera donc plus conditionnée à la complétion de l'instruction précédente. L'illustration ci-dessous illustre le fonctionnement en asynchrone.

### Exécution asynchrone



Dans cet exemple, l'ordre de complétion des processus n'est pas le même que l'ordre dans lequel ils ont été créés car leur exécution est faite de

(/dashboard)



RUN CELL



RUN ALL

**Programme I/O bound et CPU bound**

( I/O bound



( )

(834)

(831)

(/hub/dashboard)



VALIDATE



SAVE



Robin BIRON

(/hub/cheatsheet)



(/hub/progress)



(/typo)



(/forum/271)

On parle de programme **I/O bound** (limité par les entrées et sorties) lorsqu'il comporte des opérations d'entrées et de sorties qui déterminent principalement son temps d'exécution. Un programme qui comporte de nombreuses opérations de **lecture et d'écriture** d'informations externes (c'est-à-dire qui ne sont pas stockées au sein la RAM), telles que des saisies utilisateurs ou encore des requêtes adressées à une base de données, pourra être considéré comme I/O bound. Ces opérations vont donner lieu à de l'**attente**, souvent appelé **IOWait**, car ces données ne sont pas immédiatement accessible par le CPU. L'exécution du reste des instructions va donc être mise en pause en attendant l'obtention de ces informations.

La programmation **asynchrone** vise à combler ces temps d'attente. A titre d'exemple, lorsque l'on attend une réponse suite à une requête (opération I/O), si le programme est asynchrone, il est possible de lancer d'autres requêtes alors que le résultat de la première est toujours en attente. Il faut néanmoins rester vigilant car le "*multitasking*" permis par la programmation asynchrone peut parfois faire diminuer les performances car il peut s'avérer coûteux en temps. Pour désigner le passage d'une tâche à une autre, on fera souvent référence à la notion de "*context-switching*" : passer d'un thread ou d'un processus à un autre.

**CPU bound**

On dit qu'un ensemble d'instructions est CPU bound lorsque le temps d'exécution global est limité par les performances du CPU. Cela est notamment le cas d'un programme qui comporte de nombreux calculs matriciels. Ainsi plus le processeur sera performant, plus l'exécution du programme sera rapide.

**Concurrence et parallélisme**

Les notions de **concurrence** et **parallélisme** sont souvent confondues car très proches. Il est important de les distinguer. Pour cela il faut différencier l'exécution **concurrente** et **parallèle**.

**Parallélisme**

On parle d'exécution **parallèle** lorsque des tâches sont exécutées et progressent **simultanément** et **indépendamment**. Il peut s'agir de threads ou de processus.

**Concurrence**

On parle d'exécution **concurrente** lorsque plusieurs tâches progressent dans un même intervalle de temps mais s'exécutent en différé. Ce délai est bien



(/dashboard)



RUN CELL



RUN ALL



RUN ALL



souvent infime ce qui donne l'illusion d'une exécution simultanée.

Rappelons néanmoins qu'un cœur de CPU ne peut exécuter qu'une tâche à la fois et que c'est l'exploitation des temps de latence qui donnent l'illusion de simultanéité. A titre d'exemple, une seconde tâche peut être exécutée suite à l'exécution d'une première qui est toujours en cours de complétion.

(831)

(/hub/dashboard)



VALIDATE



SAVE



Robinson

BIRON

### Quoi choisir entre thread et processus ?

()

Il n'existe pas de règle générale pour faire un choix entre threads et processus, néanmoins souvent il sera pertinent de choisir :

- Plusieurs threads et une programmation concurrente lorsque le programme est I/O bound;
- Plusieurs processus et une programmation parallèle lorsque le programme est CPU bound.

(/hub/cheatsheet)



(/hub/progress)



(/typo)



(/forum/271)

## Multithreading et multiprocessing

Lorsque l'on fait référence au **multiprocessing**, on désigne souvent l'exécution de **plusieurs processus en simultané** tandis que pour le **multithreading** on fait référence à l'exécution de **plusieurs threads en concurrence**.

### Multithreading

Lorsque l'on fait référence au **multithreading**, on désigne dans la majorité des cas l'exécution de plusieurs threads au sein d'un même cœur. C'est donc de la programmation concurrente, qui permet entre autres d'exploiter les temps de latence. Il est important de noter que dans ce cas il ne s'agit pas d'exécution simultanée.

Lorsque que le processeur dispose de plusieurs cœurs alors les threads peuvent être répartis entre les cœurs et peuvent donc être exécutés simultanément, c'est ici alors de la programmation parallèle.

Il est important de souligner que le partage de mémoire entre threads et leur exécution en parallèle est à l'origine de certaines sécurités telles que les "locks" ou "synchronizers" qui permettent d'éviter autant faire que ce peut des erreurs. A titre d'exemple, si l'on incrémente un même nombre entre deux threads qui sont exécutés en parallèle alors il est possible de "perdre" ces incrémentations car les threads ont été exécutés simultanément. Les "synchronisations" entre threads ainsi que des "verrous" permettent en règle générale d'ôter ou au moins de limiter ces erreurs.

### Le cas Python

Python est un cas particulier en ce qui concerne le multithreading. En effet, il possède un verrou appelé GIL pour Global Interpreter Lock en anglais qui assure qu'un seul thread n'ait accès à l'interpréteur Python à la fois. Cela rejoint ainsi la gestion des erreurs d'accessibilité de mémoire évoquée plus haut.

### Multiprocessing

Lorsque l'on fait référence au **multiprocessing**, on désigne l'exécution des

(/dashboard)



RUN CELL

RUN ALL

processus en simultané. C'est ce que l'on appelle la programmation parallèle. L'exécution des processus est ainsi répartie entre les coeurs du processeur qui vont donc être exécutés séparément et simultanément.



( )



( )

(834)

(831)

## 2. Implémentation sur Python

(/hub/dashboard)



VALIDATE



SAVE

Robin  
BIRON

()

### Multithreading

(/hub/cheatsheet)



(/hub/progress)



(/typo)



(/forum/271)

Les gains perçus suite à l'utilisation de plusieurs threads sont assez limités sur Python. La librairie `multiprocessing` peut se révéler être une solution car elle permet de contourner le GIL en créant des sous processus au lieu de threads ce qui permet ainsi de bénéficier de meilleures performances en exploitant les différents coeurs d'un processeur.



Dans l'implémentation ci-dessous, on cherche tout de même à montrer le gain de temps potentiel obtenue suite à la création de threads. On montre ainsi que bien que limités par le GIL, des gains de temps peuvent tout de même être obtenues grâce à l'exploitation des temps de latence. Nous créons deux fonctions la première `name` qui demande à l'utilisateur de saisir le prénom Donna et qui retourne les prénoms Daniel et Donna. La deuxième fonction `calcul` effectue un calcul assez simple. Pour créer les threads nous utilisons la librairie `threading` et l'objet `Thread()`.

- (a) Exécutez la cellule ci-dessous et suivez les instructions de saisie.



In [ ]:

(/dashboard)

 **from threading import Thread**  **RUN ALL** 

```

import time

# Fonction name
def name():
    time.sleep(10)
    name1="Daniel"
    name2=input("Ecrivez le prénom : ")
    print(name1,"&",name2)

# Fonction calcul
def calcul(x):
    x=x**1000000
    print(int(str(x)[:2])) # On affiche Les deux premiers chiffres seulement

# Exécution en séquentiel

t1=time.time() # On démarre La mesure du temps

name()          # On exécute La fonction name

calcul(5)        # On exécute La fonction calcul

t2=time.time() # On arrête La mesure du temps

print("\nLa complétion du programme en séquentiel prend : ", t2-t1,"\n")

# Exécution en threads

th1=Thread(target=name) # Premier thread avec La fonction name

th2=Thread(target=calcul,args=(5,)) # Deuxième thread avec La fonction calcul

t1=time.time() # On démarre La mesure du temps

th1.start()      # On démarre Le thread 1

th2.start()      # On démarre Le thread 2

th1.join()       # On s'assure de La complétion du thread 1

th2.join()       # On s'assure de La complétion du thread 2

t2=time.time() # On arrête La mesure du temps

print("\nLa complétion du programme divisé en threads prend : ", t2-t1)

```



(/hub/dashboard)



(/hub/cheatsheet)



(/hub/progress)



(/typo)



(/forum/271)



Voici un exemple des résultats de temps de calculs que l'on a pu obtenir pour la complétion en séquentiel puis par threads. Les résultats pourront ici varier puisque la durée de complétion dépend d'une saisie utilisateur.

Ecrivez le prénom Donna: Donna  
 Daniel & Donna ← 1  
 10 ← 2

SEQUENTIEL

(/dashboard)



RUN CELL

La complétion du programme en séquentiel prend : 20.317721843719482

10 ← 1 → RUN ALL

Ecrivez le prénom Donna; Donna  
Daniel & Donna ← 2

THREADS



La complétion du programme divisé en threads prend : 12.182931886444092

(831)

(/hub/dashboard)



VALIDATE

On constate avec cet essai que l'on gagne environ 8 secondes grâce aux threads. En effet, on remarque que le résultat de la fonction calcul s'affiche avant le résultat de la fonction name, c'est l'exploitation du temps d'attente au sein du thread 1 qui permet de maximiser le temps de complétion. L'ordre de complétion est donc différent de ce qui est obtenu en séquentiel.

(/hub/cheatsheet)



(/hub/progress)



(/typo)



(/forum/271)

- (b) Exécutez la cellule ci-dessous pour observer la durée de complétion du programme en séquentiel.



In [ ]:



(/dashboard)

```
import wikipedia
# On définit la langue de préférence (834) (831)
wikipedia.set_lang("fr")
# On définit une liste de pages dont on souhaite obtenir le contenu
pages = ["Multithreading", "Threading", "Programmation_concurrente"]
# On définit une fonction qui retourne la première ligne de chaque page
# dont le nom apparaît dans la liste pages définie précédemment
def wikipedia_fonction(page):
    wiki = wikipedia.page(page)
    text = wiki.content
    print("\n", page, ' : ', text.split('.', 1)[0], "\n")
t1=time.time() # On démarre la mesure du temps
# Lancement en séquentiel à l'aide d'une boucle
for page in pages:
    wikipedia_fonction(page)
t2=time.time() # On arrête la mesure du temps
print (" \n Durée: ",(t2 - t1))
```

- (c) Implémentez des threads pour exécuter le programme ci-dessus. Observe-t-on un gain de temps ?

In [ ]:

# Insérez votre code

Hide solution



In [ ]:

(/dashboard)

```

# Solution
from threading import Thread

t1=time.time() # On démarre mesure du temps (834) (831)

# On crée une liste de threads, Le nombre de threads correspond au nombre de pages
threads = [Thread(target=web_page, args=(page,)) for page in pages]

# On démarre Les threads

for thread in threads:
    thread.start()

# On s'assure de la complétion des threads

for thread in threads:
    thread.join()

t2=time.time() # On arrête La mesure du temps

print ("\n Durée:", (t2 - t1))

# Le temps d'exécution est fortement réduit !
# L'affichage des pages n'est pas dans le même ordre que décrit au sein
# On ne contrôle pas l'ordre de complétion des threads

```

### Multiprocessing

Afin d'illustrer la parallélisation de différents processus sur Python nous utilisons la librairie `multiprocessing` et l'objet `Pool()`. Nous créons dans un premier temps, une fonction assez coûteuse en temps de calcul, `calcul_lourd`, puis nous exécutons en séquentiel cette fonction pour différents arguments. Nous faisons la même chose en parallélisant l'exécution sur 4 coeurs.

- (a) Exécutez la cellule ci-dessous.

**i** Le temps de d'exécution de la cellule ci-dessous est long (~ **5 minutes**).

In [ ]:

(/dashboard)




(/hub/dashboard)

(/hub/cheatsheet)

(/hub/progress)

(/typo)

(/forum/271)

```

import time
import multiprocessing
from multiprocessing import Pool
import numpy as np

# On définit la fonction calcul_lourd
def calcul_lourd(x):
    resultat = 0
    for k in range(1, 50):
        resultat += x * np.power(x, 1 / k * np.power(k, 3/2))
    return resultat

t1=time.time() # On démarre La mesure du temps

# On exécute la fonction pour différents arguments à la suite
calcul_lourd(range(1000000))
calcul_lourd(range(5000000))
calcul_lourd(range(4000000))
calcul_lourd(range(7000000))

t2=time.time() # On arrête La mesure du temps

print("La complétion du programme en séquentiel prend : ", t2-t1)

t1=time.time() # On démarre La mesure du temps




pool = Pool(4) # On crée Les processus

# On répartit L'exécution entre Les coeurs
resultat = pool.map(calcul_lourd,[range(1000000),range(5000000),range(4000000),range(7000000)])

t2=time.time() # On arrête La mesure du temps

print("\nLa complétion du programme en parallèle prend : ",t2-t1)

```

   Robin BIRON

La complétion du programme en séquentiel prend : 185.92671632766724


La complétion du programme en parallèle prend : 77.84622693061829

Voici un exemple de résultat que nous pouvons obtenir en termes de temps de calcul. Les résultats sont ici très parlants. Il vient que paralléliser permet de gagner plus de la moitié du temps de calcul. Il faut tout de même noter que la création des processus a un coût qui n'est pas "rentable" dès lors que les calculs sont légers. Pour des "petits" calculs, il aurait été ici plus rapide de rester en séquentiel.

Afin de vous entraîner à manipuler l'exécution parallèle des processus, vous devrez à partir de la fonction suivante `calcul_while` créer plusieurs processus pour accélérer le temps de complétion obtenu en séquentiel.

- (b) Exécutez la cellule ci-dessous pour observer la durée de la complétion du programme en séquentiel.

(/dashboard)

 RUN CELL  
In [ ]:

 RUN ALL


(/hub/dashboard)



(/hub/cheatsheet)



(/hub/progress)



(/typo)



(/forum/271)

```
#(définit )a fonction calcul_while(while ) (834) (831)
def calcul_while(x):
    while x>0:
        x=x-1
    # On définit la liste d'arguments
    nombres=[5000000,3000000,6000000,4000000,320000000,2000000, 50000000,100
    t1=time.time() # On démarre la mesure du temps
    # On exécute en séquentiel
    for nb in nombres:
        calcul_while(nb)
    t2=time.time() # On arrête la mesure du temps
    print("Durée en séquentiel : ",t2-t1)
```

- (c) Implémentez plusieurs processus pour exécuter le programme ci-dessus. Observez-vous un gain de temps ?

In [ ]:

# Insérez votre code

Hide solution



In [ ]:

(/dashboard)

# Solution n°1  
RUN CELL

RUN ALL



t1=time.time() # On démarre La mesure du temps



p1(= Pool()) # On crée Les processus )

(834)

(831)

(/hub/dashboard)

# On répartit L'exécution entre Les coeurs ()



résultat = pool.map(calcul\_les\_nombres)



SAVE



Robin

biron

t2=time.time() # On arrête La mesure du temps

(/hub/cheatsheet)

print("Durée de la solution 1 : ",t2-t1)



# Solution n°2

(/hub/progress)

# La syntaxe de La classe Process ressemble en tous points à celle de La  
# de La librairie threading

from multiprocessing import Process

(/typo)

# On crée Les processus

processes = [Process(target=calcul\_while, args=(nb,)) for nb in nombres]



(/forum/271)

t1=time.time() # On démarre La mesure du temps

if \_\_name\_\_ == '\_\_main\_\_':

# On démarre Les threads

for process in processes:  
process.start()

# On s'assure de La complétion des threads

for process in processes:  
process.join()

t2=time.time() # On arrête La mesure du temps

print("Durée de la solution 2 : ",t2-t1)

# Les durées sont à peu près équivalentes entre Les solutions.

# La classe Process et La classe Pool ne sont pas équivalentes.

# L'utilisation de La classe Pool sera plus appropriée à un grand nombre  
# qui seront réparties entre Les coeurs (nombre spécifié).

# La classe Process est plus appropriée à un nombre restreint de tâches.

# La durée de complétion est bien inférieure à celle obtenu avec une exé



### 3. Conclusion

Le **multithreading** et le **multiprocessing** sont souvent des notions  
confondues bien que dans les faits bien différentes. L'exécution parallèle de

(/dashboard)

(/hub/dashboard)

(/hub/cheatsheet)

(/hub/progress)

(/typo)

(/forum/271)



RUN CELL

RUN ALL



VALIDATE



SAVE



BIRON



Validate

compréhensibles, bien que dans les faits, bien différentes. L'exécution parallèle de threads est possible en règle générale bien que délicate compte tenu du partage de mémoire. Sur Python il n'est néanmoins possible d'exécuter qu'un seul thread à la fois à cause du GIL. Ainsi, lorsque l'on fait référence au multithreading, il s'agit la plupart du temps de programmation **concurrente** qui permet donc d'exploiter les **temps de latence** tandis que lorsque l'on fait référence au **multiprocessing** il s'agit de programmation **parallèle** qui permet de réaliser l'exécution de plusieurs processus en simultané.

(831)

