

[\(/dashboard\)](#) RUN CELL RUN ALL

(834)

(831)

[\(/hub/dashboard\)](#) VALIDATE SAVERobin  
BIRON

()

DataScientest • com

[\(/hub/cheatsheet\)](#)[\(/hub/progress\)](#)[\(/typo\)](#)[\(/forum/271\)](#)

# Python avancé

## Fonctions asynchrones

### 1. Rappel



(/dashboard)



RUN CELL

Définition

RUN ALL



(/hub/dashboard)



VALIDATE



SAVE



Notebook

(/hub/cheatsheet)



(/hub/progress)



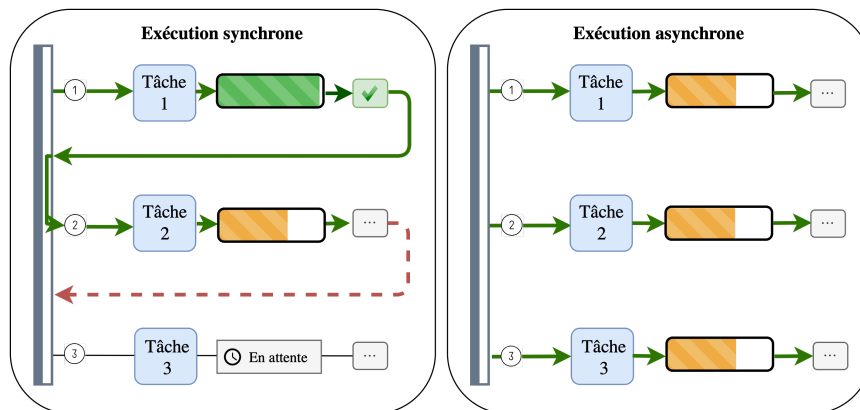
(/typo)



(/forum/271)

Un programme exécuté en asynchrone, tout comme un programme synchrone, débute l'exécution des tâches qui le composent **successivement**. La différence réside dans l'**attente de la complétion** de chaque tâche. Tandis qu'un programme synchrone attend que l'exécution de chaque tâche soit complétée avant d'exécuter la tâche suivante, un programme **asynchrone** débute l'exécution des tâches les unes à la suite des autres sans attendre. La notion de programmation **concurrente** est ainsi au coeur de la programmation asynchrone, puisque les tâches sont exécutées au sein d'un même intervalle de temps et non pas de façon séquentielle. L'exemple ci-dessous illustre le traitement synchrone et asynchrone. Pour l'exécution synchrone suite à la complétion de la tâche 1, la tâche 2 est exécutée pendant que la tâche 3 attend la complétion de la tâche 2. Pour l'exécution asynchrone, les 3 tâches sont exécutées séquentiellement mais leur exécution ne dépend pas de la complétion de la tâche précédente.

(831)



### Quelle utilité?

La programmation asynchrone permet d'accélérer le temps d'exécution et également d'augmenter la réactivité d'un programme puisqu'elle le rend plus adaptatif: on parle de scalabilité.

## 2. La programmation asynchrone sur Python



La gestion de la programmation asynchrone sur Python est particulière puisqu'elle demande de prendre en compte le GIL. Global Interpreter Lock en

(/dashboard)



RUN CELL

RUN ALL



( )



( )

(834)

(831)

(/hub/dashboard)



VALIDATE



L'avis de



Robin BIRON



(/hub/cheatsheet)



(/hub/progress)



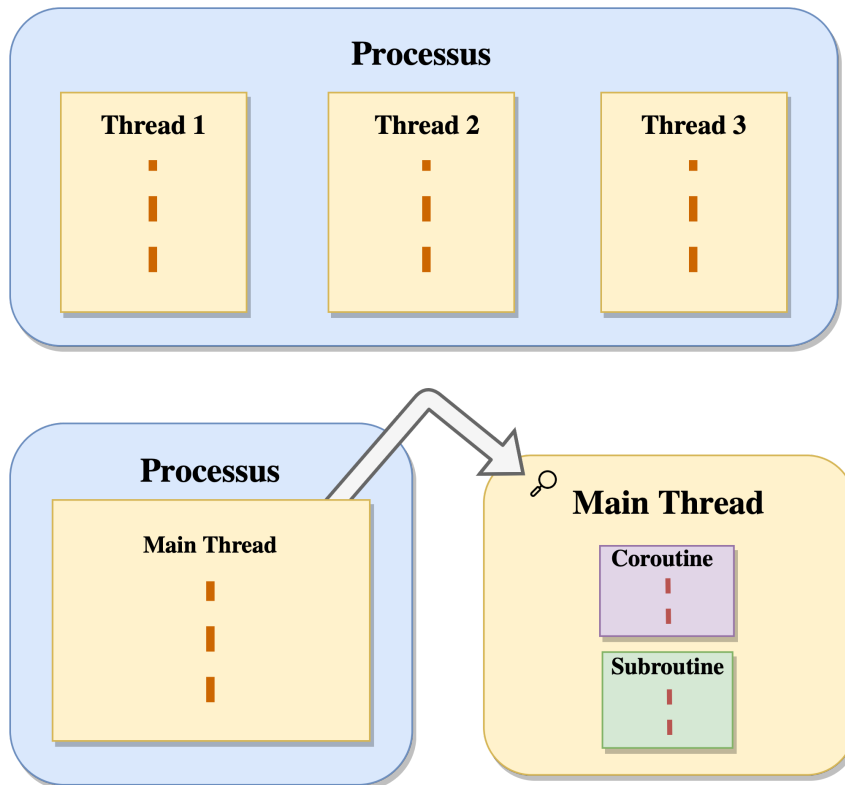
(/typo)



(/forum/271)

anglais, verrou qui assure un accès unique à l'interpréteur Python pour une question de gestion de mémoire. En effet, puisque les threads qui composent un même processus partagent la même mémoire, ce verrou permet d'éviter les erreurs générées par ces accès concurrentiels.

Bien que la création de plusieurs threads soit une des façons les plus répandues de gérer la programmation asynchrone, il existe d'autres moyens d'y parvenir. L'utilisation de routines avec une syntaxe **async/await** est une des solutions privilégiée. Il est important de noter qu'une coroutine appartient à un thread et qu'un thread appartient à un processus comme l'illustre l'exemple ci-dessous.



### Pourquoi utiliser des coroutines plutôt que des threads?

Les coroutines permettent de réaliser de la programmation concurrente en minimisant les temps alloués au "*context-switching*" induit par la présence de plusieurs threads. Il sera moins coûteux en mémoire et en temps de créer plusieurs coroutines plutôt que plusieurs threads. Les coroutines ont aussi l'avantage d'être gérées par l'utilisateur tandis que les threads le sont par le système d'exploitation. Il est tout de même à noter que les coroutines sont restreintes à la programmation concurrente et ne peuvent pas, au contraire des threads, être exécutées en parallèle.

### Coroutines

On utilise le terme coroutine pour définir à la fois les fonctions coroutines et les objets coroutines.

Les coroutines sont des fonctions qui retournent des objets coroutines. La documentation Python définit les coroutines comme une **forme généralisée**

(/dashboard)

RUN CELL

RUN ALL



de fonction. Ainsi tandis qu'une fonction standard commence son exécution en un point et termine par un autre, les coroutines possèdent plusieurs points d'entrées et de sorties et peuvent également arrêter puis reprendre leur exécution. Les coroutines sont implémentées en utilisant l'instruction `async def` et possèdent leur propre syntaxe.

(834)

(831)

(/hub/dashboard)

VALIDATE

SAVE

REVIEW

Les fonctions standards en Python sont également appelées sous-routines ou procédures. Elles sont définies par l'instruction `def`. La syntaxe d'une fonction est celle-ci:

```
def nom_de_la_fonction(liste_de_paramètres):
    instruction_n°1
    instruction_n°2
    ...
    instruction_n°k
```

Les paramètres ou encore arguments d'une fonction sont spécifiés au sein de parenthèses, il peut y en avoir un ou plusieurs comme aucun. Une fonction peut également prendre comme argument une autre fonction, ces fonctions sont appelées fonction de rappel ou "*callback*" en anglais. Les instructions correspondent au corps de la fonction et sont indentées.

On dira que l'on **appelle** une fonction lorsque l'on demande son exécution en écrivant le nom de la fonction suivi de parenthèses comprenant ses paramètres comme suit :

```
nom_de_la_fonction(liste_de_paramètres)
```

Nous allons introduire la librairie `asyncio` qui a pour vocation de faciliter la mise en place de la programmation concurrentielle sur Python grâce à la syntaxe ***async/await***.

Afin de mieux comprendre le fonctionnement de la librairie il est important de définir ce qu'est un objet **attendable**, ou *awaitable* en anglais. On parle d'objet attendable pour désigner des objets qui sont utilisables avec l'expression *await*. Les coroutines sont des attendables mais il existe deux autres sortes principales sortes d'attendables : les tâches et les futures. Tandis que les tâches, *tasks* en anglais, permettent la planification des coroutines et ainsi leur exécution concurrente, les futures désignent les résultats d'une opération asynchrone, telle que le résultat d'une coroutine. Au sein de ce notebook nous verrons les implémentations pour des coroutines et des tâches.

Puisque cela reste très abstrait, nous allons passer en revue différentes implémentations de coroutines de la librairie `asyncio` afin de concrétiser leurs utilisations. Nous implémentons dans un premier temps une coroutine très simple qui affiche la phrase "**Nous essayons les coroutines.**".

- (a) Exécutez la cellule ci-dessous pour créer la coroutine exemple.

(/dashboard)

RUN CELL

RUN ALL



de fonction. Ainsi tandis qu'une fonction standard commence son exécution en un point et termine par un autre, les coroutines possèdent plusieurs points d'entrées et de sorties et peuvent également arrêter puis reprendre leur exécution. Les coroutines sont implémentées en utilisant l'instruction `async def` et possèdent leur propre syntaxe.

(834)

(831)

(/hub/dashboard)

VALIDATE

SAVE

REVIEW

Les fonctions standards en Python sont également appelées sous-routines ou procédures. Elles sont définies par l'instruction `def`. La syntaxe d'une fonction est celle-ci:

```
def nom_de_la_fonction(liste_de_paramètres):
    instruction_n°1
    instruction_n°2
    ...
    instruction_n°k
```

Les paramètres ou encore arguments d'une fonction sont spécifiés au sein de parenthèses, il peut y en avoir un ou plusieurs comme aucun. Une fonction peut également prendre comme argument une autre fonction, ces fonctions sont appelées fonction de rappel ou "*callback*" en anglais. Les instructions correspondent au corps de la fonction et sont indentées.

On dira que l'on **appelle** une fonction lorsque l'on demande son exécution en écrivant le nom de la fonction suivi de parenthèses comprenant ses paramètres comme suit :

```
nom_de_la_fonction(liste_de_paramètres)
```




Nous allons introduire la librairie `asyncio` qui a pour vocation de faciliter la mise en place de la programmation concurrentielle sur Python grâce à la syntaxe ***async/await***.

Afin de mieux comprendre le fonctionnement de la librairie il est important de définir ce qu'est un objet **attendable**, ou *awaitable* en anglais. On parle d'objet attendable pour désigner des objets qui sont utilisables avec l'expression *await*. Les coroutines sont des attendables mais il existe deux autres sortes principales sortes d'attendables : les tâches et les futures. Tandis que les tâches, *tasks* en anglais, permettent la planification des coroutines et ainsi leur exécution concurrente, les futures désignent les résultats d'une opération asynchrone, telle que le résultat d'une coroutine. Au sein de ce notebook nous verrons les implémentations pour des coroutines et des tâches.

Puisque cela reste très abstrait, nous allons passer en revue différentes implémentations de coroutines de la librairie `asyncio` afin de concrétiser leurs utilisations. Nous implémentons dans un premier temps une coroutine très simple qui affiche la phrase "**Nous essayons les coroutines.**".



- (a) Exécutez la cellule ci-dessous pour créer la coroutine exemple.

(/dashboard)

 RUN CELL  RUN ALL 




# On importe la librairie `asyncio` et la librairie `time`

```
import asyncio,time
```

  (834) (831)

# On crée notre première coroutine

```
async def exemple():
```

 VALIDATE  SAVE  Robin BIRON

# On initialise la coroutine avec `async def`

```
print("Nous essayons les coroutines.")
```

(/hub/dashboard)

(/hub/cheatsheet)



(/hub/progress)



(/typo)



(/forum/271)

Les coroutines ne peuvent pas être appelées de la même façon que les fonctions standards sur Python. Lorsque l'on appelle une coroutine cela retourne un objet coroutine.

- (b) Exécutez la cellule ci-dessous pour observer l'objet retourné.

In [ ]:

```
# On appelle la coroutine de la même façon qu'une fonction standard
exemple()
```

**i** Une **boucle d'événements ou event loop** correspond selon la documentation Python au centre d'exécution fourni par `asyncio`. Au travers de la boucle d'événements, les tâches asynchrones sont programmées et exécutées par itération. C'est donc grâce à cette boucle d'événements que l'exécution concurrentielle est permise: chaque coroutine est exécutée jusqu'à ce que l'instruction `await` suivi d'un attendable est rencontrée où un résultat potentiel est attendu. Il est alors permis d'utiliser ce temps d'attente pour exécuter autre chose au sein de la boucle d'événements, ce qui permet de rentabiliser ce temps d'attente.



Event Loop

(/dashboard)

RUN CELL

RUN ALL



(/hub/dashboard)

VALIDATE



(/hub/cheatsheet)



(/hub/progress)



(/typo)



(/forum/271)

## Exécution concurrente

Robin  
BIRONCoroutine 1  
Tâche 1  
Tâche 2  
Await  
Tâche 3

(834)

Coroutine 2  
Tâche 1  
Tâche 2

(831)

Une coroutine est appelée au travers d'une event loop. La fonction `run()` est destinée à l'exécution des coroutines. La syntaxe est celle-ci:

```
asyncio.run(exemple())
```

```
[>>> import asyncio
[>>> async def exemple():
[...     print("Nous essayons les coroutines.")
[...
[>>> asyncio.run(exemple())
Nous essayons les coroutines.
```

Puisque nous travaillons sur Jupyter Notebook, la syntaxe y est quelque peu différente. En effet, une event loop est déjà en cours d'exécution et cela est propre à Jupyter. Pour obtenir des résultats identiques au sein d'un notebook Jupyter il suffit d'utiliser une autre syntaxe pour appeler la coroutine. Il convient d'appeler la coroutine à l'aide de `await`.

- (c) Exécutez la cellule suivante pour obtenir la sortie de la coroutine.

In [ ]:

```
# On appelle la coroutine
```

```
await exemple()
```

**i** Toutes les cellules de code qui suivent auront une durée de complétion assez longue (~ 1 minute).

Considérons la fonction `sleep` de la librairie `time` et la coroutine `sleep` de la librairie `asyncio` tandis que la première est **bloquante** la deuxième est

(/dashboard)



RUN CELL

RUN ALL



la fonction asynchrone, tandis que la première est **bloquante**, la deuxième est **non bloquante**. Cela signifie que ce temps d'attente pourra être exploité pour la réalisation d'autres tâches.



Dans la cellule ci-dessous nous implémentons une fonction qui prend en argument un prénom et qui affiche après 10 secondes d'attente. Dans la première partie de la cellule nous utilisons des fonctions standards et dans la deuxième des coroutines. Au sein de la coroutine principale main nous appelons la coroutine normale nc(prénom) avec différents arguments au sein de la coroutine gather qui permet l'exécution concurrente des coroutines.

(831)

(/hub/dashboard)



VALIDATE



Appelons la coroutine normale



Robin  
BIRON

(/hub/cheatsheet)



- (d) Exécutez la cellule suivante pour comparer la durée de complétion du code entre fonction et coroutine.

(/hub/progress)



(/typo)



(/forum/271)



In [ ]:

(/dashboard)

# Définition de la fonction nom qui en retour affiche le prénom donné en

```
def nom(prenom):
    name=prenom
    time.sleep(10)
    print(name)
```

(834) (831)

(/hub/dashboard)

# Définition de la fonction nom qui en retour affiche le temps de complétion de la fonction nom pour trois arguments différents



(/hub/cheatsheet)

```
def main():
    print("Subroutine :")
    start_time = time.time()
    nom('Daniel')
    nom('Donna')
    nom('Diane')
    end_time = time.time()
    print("Durée totale d'exécution: %.2f secondes" % (end_time - start_time))

main()
```

(/hub/progress)



(/typo)

# Définition de la coroutine nom qui en retour affiche le prénom donné en

```
async def nom_async(prenom):
    name=prenom
    await asyncio.sleep(10)
    print(name)
```

# Définition de la coroutine main qui affiche le temps de complétion de la fonction nom pour trois arguments différents

```
async def main():
    print("\nCoroutine :")
    start_time = time.time()
    await asyncio.gather(nom_async('Daniel'), nom_async('Donna'), nom_async('Diane'))
    end_time = time.time()
    print("Durée totale d'exécution: %.2f secondes" % (end_time - start_time))

await main()
```

# On constate que le temps d'exécution est divisé par trois.



Il est également possible d'exécuter de façon concurrentielle plusieurs coroutines en créant des **tâches** comme énoncé plus haut. La syntaxe est décrite dans la cellule suivante. Nous définissons deux fonctions: une fonction nom et une fonction calcul. La première affiche le prénom Daniel après un temps d'attente de 10 secondes et la seconde affiche les deux premiers chiffres d'un calcul. Nous procédons comme pour la précédente cellule nous affichons les résultats pour une fonction main standard et pour une coroutine main. Nous exécutons la fonction nom une unique fois et la fonction calcul pour deux arguments différents.

- (e) Exécutez la cellule suivante pour comparer la durée de complétion du code entre fonction et coroutine.



(/dashboard)

 RUN CELL

 RUN ALL



(

)



(

)

(834)

(831)

(/hub/dashboard)



 VALIDATE

 SAVE



Robin  
BIRON

()

(/hub/cheatsheet)



(/hub/progress)



(/typo)



(/forum/271)



In [ ]:

(/dashboard)

**# Définition de la fonction nom qui en retour affiche Le prénom Daniel**

**def nom():**  
 name="Daniel"  
 time.sleep(10)  
 print(name)

(/hub/dashboard)

**# Définition de la fonction calcul qui en retour affiche Les deux premieres lettres de la fonction nom**

**def calcul(x):**  
 x=x\*\*1000000  
 y=int(str(x)[:2])  
 print(y)

(/hub/cheatsheet)

(/hub/progress)

(/typo)

(/forum/271)

**# Définition de la fonction main qui affiche Le temps de complétion de la fonction nom et de la fonction calcul pour deux arguments différents**

**def main():**  
 print("Subroutine :")  
 start\_time = time.time()  
 nom()  
 calcul(5)  
 calcul(3)  
 end\_time = time.time()  
 print("Durée totale d'exécution: %.2f secondes" % (end\_time - start\_time))

main()

**# Définition de la coroutine nom\_async qui en retour affiche Le prénom Daniel**

**async def nom\_async():**  
 name="Daniel"  
 await asyncio.sleep(10)  
 print(name)

**# Définition de la coroutine calcul\_async qui en retour affiche Les deux premieres lettres de la fonction nom**

**async def calcul\_async(x):**  
 x=x\*\*1000000  
 y=int(str(x)[:2])  
 print(y)

**# Définition de la fonction main qui affiche Le temps de complétion de la fonction nom et de la fonction calcul pour deux arguments différents**  
**# On crée une tâche pour l'exécution de chaque fonction, cela permet de les exécuter en parallèle**

**async def main():**  
 print("\nCoroutine :")  
 start\_time = time.time()  
 task1= asyncio.create\_task(nom\_async())  
 task2= asyncio.create\_task(calcul\_async(5))  
 task3= asyncio.create\_task(calcul\_async(3))

**await task1**  
**await task2**  
**await task3**

**end\_time = time.time()**  
**print("Durée totale d'exécution: %.2f secondes" % (end\_time - start\_time))**

await main()

(/dashboard)

On constate que le temps d'exécution est divisé par deux.



( )



( )

(834)

(831)

(/hub/dashboard)



VALIDATE



SAVE



POUR BIRON

(/hub/cheatsheet)



(/hub/progress)

### 3. Exercice d'application



(/typo)



(/forum/271)

A partir du code dans la cellule ci-dessous, implémentez des coroutines qui permettent d'obtenir un temps d'exécution inférieur à celui obtenu avec une programmation synchrone et des fonctions standards. A l'aide de l'API wikipédia, nous souhaitons résumer les pages concernant les coroutines, le threading et la programmation concurrente. Pour cela, nous implémentons une fonction `wiki` qui affiche en retour le résumé d'une page wikipédia, l'argument de la fonction. Nous ajoutons au sein de la fonction un temps d'attente.

- (f) Exécutez la cellule suivante puis dans la prochaine cellule insérez votre code.



In [ ]:

(/dashboard)



(/hub/dashboard)



(/hub/cheatsheet)



(/hub/progress)



(/typo)



(/forum/271)



# On importe la librairie wikipedia

RUN CELL



RUN ALL



import asyncio,wikipedia,time

# On définit la langue de préférence (834) (831)

wikipedia.set\_lang("fr")

()



# On définit une fonction



SAVE



Robin

Biron

return Le résumé de chaque page et qui a

def wiki(page):

time.sleep(6)

print('\n',wikipedia.summary(page))

# On définit une fonction main pour lancer la fonction avec différents a

def main():

print("Subroutine :")

start\_time=time.time() # On démarre la mesure du temps

wiki("Coroutine")

wiki("Threading")

wiki("Programmation\_concurrente")

end\_time=time.time() # On arrête la mesure du temps

print("\nDurée totale d'exécution: %.2f secondes" % (end\_time - start\_time))

# On appelle la fonction main

main()

In [ ]:

# Insérez votre code

import asyncio,wikipedia,time

wikipedia.set\_lang("fr")



Hide solution



In [ ]:

(/dashboard)



# On définit une coroutine qui retourne le résumé de chaque page et qui

**import** asyncio,wikipedia,time

wikipedia.set\_lang("fr")

( )

(834)

(831)

(/hub/dashboard)

**async def** wiki\_async(page):**await** asyncio.sleep(6)

()

**print**(' \n',wikipedia.summary(page))

# On définit la coroutine main pour lancer les coroutines simultanément

**async def** main():**print**("Coroutine :")

start\_time=time.time() # On démarre la mesure du temps

**await** asyncio.gather(wiki\_async("Coroutine"),wiki\_async("Threading"))

end\_time=time.time() # On arrête la mesure du temps

**print**(" \nDurée totale d'exécution: %.2f secondes" % (end\_time - start\_time))

# On appelle la coroutine main

**await** main()

# On constate bien un gain de temps substantiel entre les deux méthodes.

(/hub/cheatsheet)



(/hub/progress)



(/typo)



(/forum/271)

## 4. Conclusion

Les coroutines sont des implémentations assez efficaces lorsqu'il s'agit de programmation concurrente. Elles permettent en effet d'utiliser les temps d'attente afin de démarrer d'autres instructions et donc d'accélérer le temps de complétion. Assez simples à implémenter, elles peuvent être utilisées à la place des threads ou en combinant les deux. Il faut néanmoins bien garder en tête que les coroutines ne permettent pas d'exécuter du code en parallèle comme le permettent les threads ou les processus.

**Validate**