

# Python : programmation asynchrone

## Présentation

Un **programme** est une succession d'instructions qui s'**exécutent plus ou moins rapidement** :

- si ces instructions doivent **obligatoirement s'enchaîner** parce qu'elles sont **dépendantes** les unes des autres, le **temps d'exécution de la totalité du programme** résultera de la **somme du temps d'exécution individuel** de chaque instruction.
- si certaines instructions sont **indépendantes** entre elles, elles seront cependant **bloquées** le temps que se **terminent** les instructions précédentes.

Ce mode de programmation est un appelé **programmation synchrone**.

Le mode **programmation asynchrone** permet alors de **lancer** l'exécution d'une instruction et, **sans attendre** la fin de l'exécution de celle-ci, de **lancer** l'exécution d'une autre instruction.

Cela est utile pour des instructions qui s'**exécutent lentement** comme la récupération ou l'écriture de données depuis le **réseau** ou sur le disque dur (les entrée/sorties) et que l'on souhaite ne pas **bloquer** le programme tant que ces instructions ne sont pas **terminées**. Cela concerne donc particulièrement les **applications du Web**.

La programmation **asynchrone** permet de gérer un ensemble **d'événements** de façon **concurrente**.

## Principe de la programmation asynchrone

L'idée générale de la programmation asynchrone est de pouvoir **démarrer des opérations non bloquantes** de façon **concurrente**, puis de pouvoir suivre leurs états et finalement exécuter des fonctions de retour (callbacks) quand elles sont prêtes.

Pour cela le programme va **gérer une boucle** qui attend que des **événements** (event loop) se produisent et alors fait **correspondre** à chaque événement qui survient la **fonction** qui lui est associée. C'est dans la fonction associée que sont définis les traitements à faire.

### Exemple pour un serveur Web :

- le serveur Web à un **seul point d'entrée** et dispose de **plusieurs pages web** ;
- le serveur est en **attente**, c'est à dire à **l'écoute** des requêtes des clients (les navigateurs) qui peuvent demander telle ou telle page ;
- dès qu'une **requête arrive**, le serveur Web **retourne la page correspondant à la demande**.

Précisions :

- **chaque** requête des clients est considérée comme un **événement distinct** (event),
- le serveur est programmé pour qu'à **chaque** événement une **fonction prédéfinie** soit exécutée. L'événement est le **déclencheur** de l'exécution de la fonction.

## La bibliothèque asyncio

La bibliothèque **asyncio** permet de faire de la programmation asynchrone et dispose de deux éléments fondamentaux, une **event loop** et la **classe Future**.

La **mise en oeuvre** de la programmation asynchrone avec la bibliothèque asyncio nécessite l'utilisation deux mots clés **async** et **await** :

- **async** permet de **définir la fonction comme asynchrone**,
- dans le corps de la fonction, **await** permet de préciser qu'une instruction **suspend l'exécution** de la fonction asynchrone le temps que cette instruction se **termine**. Pendant ce temps le programme qui a appelé la fonction asynchrone pourra **continuer son exécution** normalement en attendant qu'elle soit **débloquée**.

## Première manière de gérer une boucle d'événement

Première possibilité pour **créer** une boucle d'événement (event loop) :

- on **instancie** une **boucle d'événement** avec la méthode **asyncio.geteventloop()**
- on précise, durant cette boucle d'événement **quelle fonction** doit être **exécutée** avec la méthode **rununtilcomplete(maCoroutine())**. La fonction qui est exécutée dans la boucle d'événement est appelée une **coroutine**.
- on définit une fonction comme un coroutine en rajoutant le mot clé **async**.

```
import asyncio
import time

# Définir une coroutine qui sera exécutée ultérieurement (dans le futur)
async def maCoroutine():
    print("Début de ma Coroutine")
    time.sleep(2)
    print("Fin de ma Coroutine")

# Faire tourner une simple boucle d'événement
# jusqu'à ce que l'exécution de la coroutine soit finie (completed)
loop = asyncio.get_event_loop()
loop.run_until_complete(maCoroutine())

# fermer la boucle d'événement
print("fermer la boucle d'événement")
loop.close()
```

## Les options d'exécution

Pour exécuter la boucle d'événement on peut :

- appeler la méthode **rununtilcomplete(future)** et terminer la boucle d'événement quand la fonction future (la coroutine) a **terminé** son exécution,
- appeler la méthode **run\_forever()** qui exécutera la boucle d'événement jusqu'à ce que la méthode **stop()** soit appelée ou si le programme se termine.

## Autre alternative pour gérer une boucle d'événement avec la méthode run\_forever()

La méthode **runforever()** permet à la boucle d'événement de s'exécuter indéfiniment jusqu'à ce que le programme se termine ou si la méthode **stop()** est appelée. La méthode **ensurefuture()** permet également de déclarer une coroutine et sera utile pour déclarer plusieurs coroutines dans la même boucle d'événement. `<code python> import asyncio async def maCoroutine(): while True: print("Début de ma Coroutine") await asyncio.sleep(1) print("Fin de ma Coroutine") loop = asyncio.geteventloop() try: asyncio.ensurefuture(maCoroutine()) loop.runforever() except KeyboardInterrupt: pass finally: # fermer la boucle d'événement print("fermer la boucle d'événement") loop.close() </code>`

### Précisions :

La fonction utilise une **boucle infinie** et donc le programme va **boucler indéfiniment**. Il est possible alors :

- d'**interrompre** le programme avec les touches d'interruption **CTRL + C**,
- de **gérer l'exception** qui est générée,
- pour afficher **Fermer proprement** la boucle et **terminer proprement** le programme.

==== Gérer des tâches en parallèle ==== Un intérêt de la programmation asynchrone est de pouvoir gérer des tâches en parallèle :  
 \* une première tâche est **lancée**, \* une **deuxième** tâche, qui ne dépend pas de la fin de l'exécution de la première tâche peut être **lancée**, alors même que la première tâche n'est **pas terminée**. `<code python> import asyncio async def tache_1(): while True: print("Début de ma tâche 1") await asyncio.sleep(1) print("Fin de ma tâche 1") async def tache2(): while True: print("Début de ma tâche 2") await asyncio.sleep(5) print("Fin de ma tâche 2") loop = asyncio.getevent_loop() try: asyncio.ensurefuture(tache1()) asyncio.ensurefuture(tache2()) loop.runforever() except KeyboardInterrupt: pass finally: # fermer la boucle d'événement print("Fermer la boucle d'événement") loop.close() </code> Voici le résultat obtenu : <code python> >> RESTART: D:/Developpement/python/Websocket/scripts/coroutinefutureparallele.py Début de ma tâche 1 Début de ma tâche 2 Fin de ma tâche 1 Début de ma tâche 1 Fin de ma tâche 1 Début de ma tâche 1 Fin de ma tâche 1 Début de ma tâche 1 Fin de ma tâche 1 Début de ma tâche 1 Fin de ma tâche 2 Début de ma tâche 2 Fin de ma tâche 1 Début de ma tâche 1 Fin de ma tâche 1 Début de ma tâche 1 Fermer la boucle d'événement >> </code> La tâche 1 s'exécute plus souvent que la tâche 2. Ces deux tâches s'exécutent indépendamment l'une de l'autre. ==== Retour au cours : Les instructions du langage Python ====`

- [Cours : Les instructions du langage Python](#)

From:

[/ - Les cours du BTS SIO](#)

Permanent link:

[/doku.php/icn/facultatif/c\\_langage\\_python\\_asynchrone](#)

Last update: **2018/04/08 14:53**

