Programmation asynchrone en Python

Présentation

Un programme informatique réalisé avec un **mode de programmation synchrone** consiste en une **liste de différentes tâches** qui s'exécutent les **unes après les autres**. Dans cette liste, une tâche doit **attendre** que les toutes les tâches **précédents soient terminées** avant d'être exécutée. De même, toutes les tâches **successives** doivent attendre que cette tâche soit terminée pour être à leur tour exécutées.

Ces tâches sont **concurrentes** et sont considérées comme **bloquantes** pour toutes les tâches qui suivent.

Cela convient pour de nombreux programmes informatiques.

La **programmation asynchrone** permet à un programme de lancer plusieurs tâches **sans attendre** que chacune de ces tâches soient terminées avant de passer à la suivante. Cette manière de programmer permet :

- de **ne pas bloquer** le déroulement d'un programme qui attendrait la fin d'une tâche avant de passer à la suivante,
- **d'optimiser** les temps de traitement.

Pour cela il est nécessaire que les tâches le **permettent**, c'est à dire que certaines **tâches soient indépendantes de la fin de l'exécution d'autres tâches**.

Exemples de programmation asynchrone

- les applications réseaux et notamment les applications Web, où les temps de réponse entre le client et le serveur peuvent être très variables en fonction de l'état du réseau ou des délais de traitement sur le serveur.
- les accès à des données (disque dur, base de données, ressources sur le réseau, IoT, etc.) qui sont plus lents que les accès à la mémoire de l'ordinateur.
- la gestion d'un websocket, où il est nécessaire, pour le serveur tout comme le client :
  - de pouvoir envoyer des données,
  - et **EN MEME TEMPS** de recevoir des données.
  - Si on utilise un modèle de programmation synchrone, le serveur (ou le serveur) doit réaliser une tâche après l'autre car il ne peut à la fois attendre des données et en envoyer. Ou bien il se consacre alternativement à l'une ou à l'autre de ces deux tâches **concurrentes** sans savoir à priori s'il a plus de données à recevoir ou à envoyer.

Le modèle de programmation asynchrone permet alors de lancer plusieurs tâches :

- tâches qui doivent bien sûr pouvoir s'**exécuter indépendamment** l'une de l'autre,
- tâches appelée **coroutines** et qui s'exécutent l'une après l'autre, dans le désordre, de manière non bloquante,
- coroutines qui s'exécutent dans le **même processus** et qui peuvent s'**échanger des messages** et se **partager des informations**.

Le langage Python dispose depuis la version 3.6 de la bibliothèque **asyncio** pour gérer des traitements **asynchrones**.

Installer la version python3.6 :

- ajouter la ligne suivante au fichier /etc/apt/sources.list

```
$ sudo nano /etc/apt/sources.list
# add
deb http://ftp.de.debian.org/debian testing main
```

- mettre à jour la liste des paquets

```
$ sudo apt-get update
```

- installer python 3.6

```
$ sudo apt-get -t testing install python3.6
```

Principe de programmation avec la bibliothèque asyncio

liens :

- https://tutorialedge.net/python/concurrency/asyncio-event-loops-tutorial/
- https://www.artificialworlds.net/blog/2017/05/31/basic-ideas-of-python-3-asyncio-concurrency/

Le composant principal de tout programme Python basé sur la bibliothèque **asyncio** est la gestion d'une **boucle d'événement** (event loop) qui se charge de lancer les différentes tâches (**coroutines**) du programme.

La boucle d'événements attend que des événements se produisent et fait correspondre à chacun de ces événements une fonction qui a été explicitement associée à ce type d'événement.

- Instancier une boucle d'évènement : <code python> loop = asyncio.get*event*loop() </code>
- choisir les options d'exécution de la boucle d'évèvnement :
    - soit indiquer la fonction (la coroutine) à exécuter durant cette boucle et, quand la coroutine aura fini son exécution, d'arrêter la boucle d'événement :

      ```
      loop.run_until_complete(Coroutine())
      loop.close()
      ```

    - soit de faire exécuter indéfiniment la boucle d'événement jusqu'à ce que la fonction stop() soit appelée :

  <code python> # définir quelle **coroutine** doit être **exécutée** : asyncio.ensure*future(Coroutine()) loop.run*forever() loop.close() </code>

  > Attention : cette méthode va faire boucler indéfiniment la boucle d'événement. Il faut alors gérer l'arrêt de la boucle

- la définition de la coroutine se fait en utilisant le mot **clé async** :

Code avec la méthode run_until_complete()

```
import asyncio

# Définir la coroutine qui sera exécutée ultérieurement (future)
async def Coroutine():
    print("Exécution de la coroutine")

# Création de la boucle d'événement (event loop)
loop = asyncio.get_event_loop()

# exécuter la boucle d'événement jusqu'à la fin de l'exécution de la coroutine
loop.run_until_complete(Coroutine())

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

Code avec la méthode run_forever()

```
import asyncio

# Définir la coroutine qui sera exécuter ultérieurement (future)
async def Coroutine():
    print("Exécution de la coroutine")

# indiquer que la coroutine sera exécutée dans la boucle d'événement
# Celle-ci est planifiée en arrière plan mais pour l'instant
# le programme n'attend pas le résultat de son exécution
asyncio.ensure_future(Coroutine())

# Création de la boucle d'événement (event loop)
loop = asyncio.get_event_loop()

#lancer indefiniment la boucle -> bouclage infini du programme
```

```
loop.run_forever()

# fermer la boucle d'événement -> ne sera jamais exécuté en l'état
loop.close()
```

Pour **arrêter** la boucle d'événement il faut appeler la **méthode stop()**, par exemple dans la **coroutine** :

```
# Définir la coroutine qui sera exécuter ultérieurement (future)
async def Coroutine():
    print("Exécution de la coroutine")
    loop.stop()
```

Si la méthode **stop() n'est appelée** le programme va **boucler indéfiniment**.

Lancer plusieurs coroutines

Pour lancer plusieurs coroutines :

- il faut mettre chaque coroutine en file d'attente avec la méthode **asyncio.ensure_future (function ())**.
- utiliser le mot clé **await** sur une instruction de la coroutine qui est bloquante.

Le mot clé **await** sur une instruction de la coroutine qui est bloquante :

- il s'agit d'une instruction qui **prend du temps à s'exécuter**.
- le temps que cette instruction se termine, les autres instructions de la coroutine sont **mises en attente** et la **main est rendue** à la boucle d'événement, pour **lancer** d'autres tâches ou **poursuivre** l'exécution d'autres tâches.
- dès que l'instruction qui prenait du temps est **terminée**, les autres instructions de la coroutine sont **exécutées**. </WRAP>

```
import time
import asyncio

# Définir deux coroutines qui seront exécutée ultérieurement (future)
async def Coroutine_1():
    print("Exécution de la coroutine 1")
    await asyncio.sleep(2)
    print("Fin de l'exécution de la coroutine 1")

async def Coroutine_2():
    print("Exécution de la coroutine 2")
    await asyncio.sleep(2)
    print("Fin de l'exécution de la coroutine 2")
    loop.stop()

# Création de la boucle d'événement (event loop)
loop = asyncio.get_event_loop()

# indiquer les coroutines à exécuter dans la boucle d'événement
asyncio.ensure_future(Coroutine_1())
asyncio.ensure_future(Coroutine_2())

#lancer indefiniment la boucle -> bouclage infini du programme
loop.run_forever()

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

- autre manière de le gérer avec la **méthode gather()**

```
import time
import asyncio

# Définir deux coroutines qui seront exécutée ultérieurement (future)
async def Coroutine_1():
    print("Exécution de la coroutine 1")
```

```
    await asyncio.sleep(2)
    print("Fin de l'exécution de la coroutine 1")

async def Coroutine_2():
    print("Exécution de la coroutine 2")
    await asyncio.sleep(2)
    print("Fin de l'exécution de la coroutine 2")
    loop.stop()

# Création de la boucle d'événement (event loop)
loop = asyncio.get_event_loop()

# indiquer dans une variable liste_coroutines la liste de coroutines
# à ajouter à la boucle d'événement
liste_coroutines = asyncio.gather(Coroutine_1(), Coroutine_2())

#exécuter la boucle d'événement jusqu'à la fin de l'exécution des coroutines
loop.run_until_complete(liste_coroutines)

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

Appeler une coroutine depuis une autre coroutine

Dans les situations précédentes :

- Des coroutines indépendantes les unes des autres étaient tout d'abord planifiées en arrière plan en tant qu'objet coroutine sans être exécutées.
- Pour que le code d'une coroutine soit exécuté, celle-ci doit être mise dans une boucle d'événement.
- Pour que la coroutine ne bloque le programme le mot clé await précise l'instruction de la coroutine qui est prend du temps à s'exécuter afin de rendre la main au planificateur afin que celui-ci gère l'exécution d'autres tâches.

Voici comment on peut gérer l'appel de la coroutine 2 depuis la première :

```
import time
import asyncio

# Définir deux coroutines qui seront exécutée ultérieurement (future)
async def Coroutine_1():
    print("Exécution de la coroutine 1")
    await  Coroutine_2()
    print("Fin de l'exécution de la coroutine 1")
    return "1"

async def Coroutine_2():
    print("Exécution de la coroutine 2")
    await asyncio.sleep(2)
    print("Fin de l'exécution de la coroutine 2")
    return "2"
    #loop.stop()

# Création de la boucle d'événement (event loop)
loop = asyncio.get_event_loop()
print(loop.run_until_complete(Coroutine_1()))

# fermer la boucle d'événement -> est maintenant exécuté
loop.close()
```

Parties suivantes à approfondir ...

Ressources :

- https://www.artificialworlds.net/blog/2017/05/31/basic-ideas-of-python-3-asyncio-concurrency/
- https://stackoverflow.com/questions/42231161/asyncio-gather-vs-asyncio-wait
- https://docs.python.org/3/library/asyncio-task.html

Gérer différentes tâches : non finalisé

Les tâches dans Asyncio sont responsables de l'exécution des coroutines dans une boucle d'événement. Ces tâches ne peuvent s'exécuter que dans une boucle d'événement à la fois et, pour réaliser l'exécution en parallèle, vous devez

exécuter plusieurs boucles d'événements.

La fonction **wait ()** permet d'attendre jusqu'à ce que les instances Future, les tâches, soient terminée. Cette fonction renvoie alors un ensemble de 2 ensembles nommés. Le premier jeu contient les tâches terminées, le second les tâches non terminées.

```
#envoyer des messages en parallèle
    envoyer = asyncio.ensure_future(gestion_envoi_message(websocket))
    #recevoir message en parallèle
    recevoir = asyncio.ensure_future(gestion_reception_message(websocket))
    termine, attente = await asyncio.wait(
            [envoyer, recevoir],
            return_when = asyncio.FIRST_COMPLETED,
    )
```

Les activités ...

Je reviens à la liste des activités.