R308 : Consolidation de la programmation
Chapitre 4 : Les Threads en python :

1 Présentation :

Un Thread est une portion de code capable de s'exécuter en parallèle d'autres traitements.
Les Threads sont très utiles et parfois indispensables :

- traitement simultané d'instructions bloquantes (ex : écoute d'un serveur multi-clients).
- programmation événementiel et graphique.

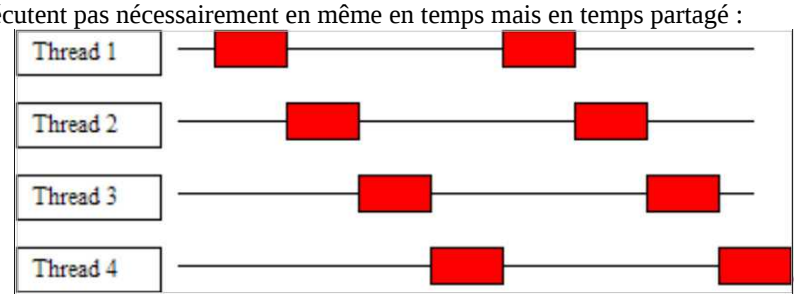
On parle de « Thread » ou « Taches » ou « processus légers »

1.1 Quelques liens

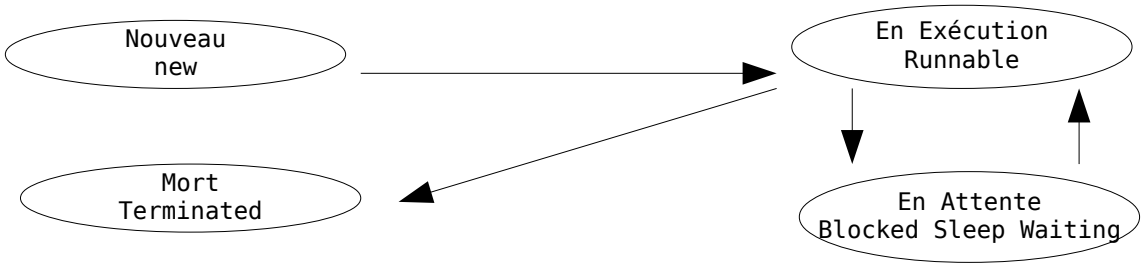
https://python.developpez.com/faq/?page=Thread
https://docs.python.org/fr/3/library/threading.html#thread-objects

1.2 Représentation schématique :

Les Threads ne s'exécutent pas nécessairement en même en temps mais en temps partagé :



1.3 Les 4 états d'un Thread :



1. État nouveau ou état initial.

   Le Thread est opérationnel (instancié) mais n'est pas encore actif.
2. L'exécution

   Elle est lancée par la méthode start()
3. En attente : le Thread n'exécute aucun traitement et ne consomme aucune ressource
   - Appel de la méthode Thread.sleep(temps en millisecondes).
   - Accès à une ressource bloquante (flux d'entrée, accès en base de données, etc…)
4. État mort :
   - Lorsque le tache est terminée.
   - Lorsqu'une erreur s'est produite (ex : flux réseau coupé).
   - Lorsqu'il est tué (à éviter fortement)
5. Quelques remarques :
   - La fonction main() correspond au Thread principal.
   - Une fois mort, un Thread ne peut pas être à nouveau exécuté. Il sera nécessaire de le ré-instancier
   - Les Threads possèdent tous une méthode run() qui est exécutée à partir de la méthode start()
   - La méthode run() d'un Thread ne peut avoir ni paramètre ni retour.
6. Il existe plusieurs possibilités pour exécuter une suite d'instruction dans un Thread :
   - En ciblant directement une fonction ou une méthode
   - Par héritage de la classe Thread du module threading

1.4 Les Threads en python :

Les Thread en Python s'utilisent relativement facilement.
Pour cela, il faut déclarer une instance en utilisant le constructeur :

```
threading.Thread( group=None, target=None, name=None, args=(), kwargs={}) où :
```

- group doit rester à None, en attendant que la classe ThreadGroup soit implantée.
- **target** est la fonction ou méthode ciblée par le Thread.
- name est le nom du Thread.
- **args** est un tuple d'arguments pour l'invocation de la fonction target
- kwargs est un dictionnaire d'arguments pour l'invocation de la fonction target

1.5 Premier exemples : un thread cible une fonction :

1. Soit la fonction ci dessous :

La fonction permet d'afficher la valeur d'un compteur entre 0 et nb_maxi avec une pause aléatoire entre chaque nouvel affichage.

```
from threading import Thread, current_thread
from random import random
import time

def fonction_compteur(nb_maxi: int)->None:
      nom:str = current_thread().name # (1)
      compteur:int = 0
       pause:float
      while compteur < nb_maxi:
            pause = 0.5 + 0.5 * random() # pause aléatoire entre 0.5 et 1 seconde
            print(f"{nom}: {compteur} pause: {pause:.2f} s")
            time.sleep(pause)
            compteur += 1
      print(nom + " a terminé de compter")
```

(1) threading.current_thread().getName() permet de récupérer le nom du thread courant

Le programme principal lance 2 threads simultanément :

```
# Thread et fonctions
print("exécution d'une fonction dans un Thread")
# declaration de 2 threads
th1: Thread = None
th2: Thread = None
# instanciation des threads
th1 = Thread(target=fonction_compteur, name = "f1", args=(10,)) # (1)
th2 = Thread(target=fonction_compteur, name = "f2", args=(4,)) #(1)

# exécution des Threads
th1.start()
th2.start()
```

(1) : target : nom de la fonction (… pas de parenthèses)
args est un tuple : la virgule est obligatoire s'il n'y a qu'un seul paramètre

Résultat :

```
f1 : 0 pause : 0.92 s
f2 : 0 pause : 0.65 s
f2 : 1 pause : 0.53 s
f1 : 1 pause : 0.92 s
f2 : 2 pause : 0.87 s
f1 : 2 pause : 0.57 s
f2 : 3 pause : 0.77 s
f1 : 3 pause : 0.65 s
f2 a terminé de compter
f1 : 4 pause : 0.90 s
f1 : 5 pause : 0.60 s
f1 : 6 pause : 0.64 s
f1 : 7 pause : 0.95 s
f1 : 8 pause : 0.99 s
f1 : 9 pause : 0.80 s
f1 a terminé de compter
```

1.6 Deuxième exemple : un thread cible une méthode :

Remarque : la fonction précédente est simplement transférée dans une classe

```
class Classe_compteur:
      def compte(self, nb_maxi: int):
            nom:str = nom:str = current_thread().name # (1)
            compteur:int = 0
             pause:float
            while compteur < nb_maxi:
                  pause = 0.5 + 0.5 * random() # pause aléatoire entre 0.5 et 1 seconde
                  print(f"{nom}: {compteur} pause: {pause:.2f} s")
                  time.sleep(pause)
                  compteur += 1
            print(nom + " a terminé de compter")
```

Programme principal :

```
print("exécution d'une méthode dans un Thread avec passage de paramètres")
c1: Compteur = None
c2: Compteur = None

c1 = Compteur()
c2 = Compteur()

th1: Thread = None
th2: Thread = None

th1 = Thread(target=c1.compte, name = "c1", args=(3,))
th2 = Thread(target=c2.compte, name = "c2", args=(5,))

th2.start()
th1.start()
```

Résultat :

```
c2 : 0 pause : 0.94 s
c1 : 0 pause : 0.92 s
c1 : 1 pause : 0.94 s
c2 : 1 pause : 0.55 s
c2 : 2 pause : 0.52 s
c1 : 2 pause : 0.53 s
c2 : 3 pause : 0.86 s
c1 a terminé de compter
c2 : 4 pause : 0.83 s
c2 a terminé de compter
```

1.7 Troisième exemples : un thread cible une méthode, utilisation du constructeur pour le passage des paramètres de l'objet :

Cette façon de faire est très pratique

```
class Classe_compteur1:
      def __init__(self, nom: str, nb_maxi: int):
            self.__nom:str = nom
            self.__nb_maxi:int = nb_maxi
      def compte(self)-> None:
            compteur:int = 0
            pause:float
            while compteur <self.__nb_maxi:
                  pause = 0.5 + 0.5 * random() # pause aléatoire entre 0.5 et 1 seconde
                  print(f"{self.__nom}: {compteur} pause: {pause:.2f} s")
                  time.sleep(pause)
                  compteur += 1
      print(self.__nom + " a terminé de compter")
```

Remarque :

On utilise le constructeur pour passer les paramètres nécessaires.

La méthode compte() n'a pas de paramètre

Programme principal :

```
print("exécution d'une méthode dans un Thread avec passage de paramètres dans le constructeur")
c1: Compteur1 = None
c2: Compteur1 = None
th1: Thread = None
th2: Thread = None
c1 = Compteur1("c1", 10)
c2 = Compteur1("c2", 5)
th1 = Thread(target=c1.compte)
th2 = Thread(target=c2.compte)
th2.start()
th1.start()
```

1.8 Quatrième exemple : Utilisation de l'héritage pour la mise en œuvre des Threads :

La classe compteur hérite de la classe Thread

```
class Classe_compteur_heritage(Thread):
      def __init__(self, nom: str, nb_maxi: int):
            Thread.__init__(self, name = nom) # appel au constructeur de la classe mère
            self.__nb_maxi:int = nb_maxi
      # surdéfinition de la méthode run() de la classe Thread
      def run(self)-> None:
            nom:str = super().name # (1)
            compteur:int = 0
            pause:float
            while compteur <self.__nb_maxi:
                  pause = 0.5 + 0.5 * random() # pause aléatoire entre 0.5 et 1 seconde
                  print(f"{nom}: {compteur} pause: {pause:.2f} s")
                  time.sleep(pause)
                  compteur += 1
            print(nom + " a terminé de compter")
            # (1) getName() est une méthode de la classe Thread
```

(1) getName() est une méthode de la classe Thread

Remarques :

La méthode run() est obligatoire.

La méthode run() n'a ni paramètre ni retour.

Programme principal :

```
print("Thread et héritage")
# déclaration des références d'objets
c1: CompteurHeritage = None
c2: CompteurHeritage = None
# instanciation des objets
c1 = CompteurHeritage("c1", 10)
c2 = CompteurHeritage("c2", 5)
# exécution des Threads
c1.start()
c2.start()
```

1.9 Quelques méthodes de la classe Thread :

Thread() constructeur	Thread(target=None, name=None, args=())
start()	Lance l'activité du fil d'exécution.
run()	Méthode représentant l'activité du fil d'exécution.
join(timeout=None)	Attend que le fil d'exécution se termine. Ceci bloque le fil appelant jusqu'à ce que le fil dont la méthode join() est appelée se termine ou jusqu'à ce que le délai optionnel timeout soit atteint.
is_alive()	Renvoie si le fil d'exécution est vivant ou pas (un booléen)

1.10 ACCÈS CONCURRENT : Protection d'une portion de code par l'utilisation d'un verrou :

1. Présentation :

Il peut arriver que plusieurs threads modifient les mêmes données.
Il y a alors un risque de conflit.
Il est possible en python de protéger une portion de code à l'aide d'un verrou
Le verrou doit être UNIQUE et connu de tous les threads concernés

2. Implémentation en python :

La classe Lock du module threading permet d'implémenter les objets de verrouillage. Une fois qu'un thread a acquis un verrou, les tentatives suivantes pour l'acquérir se bloquent jusqu'à ce qu'il soit libéré; n'importe quel fil peut le libérer.

- La méthode acquire(blocking=True, timeout=- 1) permet d'acquérir un verrou
- La méthode release() permet de relâcher le verrou. La méthode peut être appelée par n'importe quel thread.
- La méthode locked() retourne True si le verrou est activé.

3. Implémentation à partir de l'exemple précédent : le verrou est un paramètre du constructeur

```
from threading import Thread, current_thread, Lock

class CompteurHeritage(Thread):
      def __init__(self, nom:str, nb_maxi:int, verrou:Lock):
            Thread.__init__(self, name=nom) # appel
            self.__nb_maxi:int = nb_maxi
            self.__verrou:Lock = verrou
      # surdéfinition de la méthode run() de la classe Thread
      def run(self)-> None:
            nom:str = current_thread().name
            compteur:int = 0
            pause:float
            self.__verrou.acquire() # debut portion de code à protéger
            while compteur < self.__nb_maxi:
                  pause = 0.5 + 0.5 * random() # pause aléatoire entre 0.5 et 1 seconde
                  print(f"{nom}:{compteur} pause:{pause:.2f} s")
                  time.sleep(pause)
                  compteur += 1
            print(nom + " a terminé de compter")
            self.__verrou.release() # fin de la portion de code à protéger
```

```
# declaration des variables
c1:CompteurHeritage = None
c2:CompteurHeritage = None
verrou:Lock = None
# instantiation des variables
verrou = Lock()
c1 = CompteurHeritage("c1", 10,verrou)
c2 = CompteurHeritage("c2", 5, verrou)
# exécution des Thread
c1.start()
c2.start()
```

4. Seconde implémentation en python :

Il sera également possible de définir le verrou comme étant un attribut statique de la classe Compteur.
Le verrou sera donc partagé par toutes les instances.

```
class Compteur_heritage(Thread):
    VERROU:Lock = Lock() # variable statique
    def __init__(self, nom:str, nb_maxi:int)->None:
        Thread.__init__(self, name = nom) # (1)
        self.__nb_maxi = nb_maxi
    # surdéfinition de la méthode run() de la classe Thread
    def run(self)-> None:
        Compteur_heritage.VERROU.acquire() # debut portion de code à protéger
        ...
        Compteur_heritage.VERROU.releaserelease() # fin de la portion de code à protéger
```

(1) : le verrou n'est pas un paramètre du constructeur.

5. Utilisation de l'instruction with :

```
      def run(self)-> None:
            nom:str = current_thread().name
            whith VERROU:# debut portion de code à protéger
                  compteur:int = 0
                  pause:float
                  while compteur < self.__nb_maxi:
                        pause = 0.5 + 0.5 * random() # pause
                        print(f"{nom}:{compteur} pause:{pause:.2f} s")
                        time.sleep(pause)
                        compteur += 1
                  print(nom + " a terminé de compter")
            # fin de la portion de code à protéger
```