NOHAR

Zeste de savoir

Découvrons la programmation asynchrone en Python

2 septembre 2022

Table des matières

Introduction

Depuis que Python 3.5 est sorti, un nom se trouve sur les lèvres de tous les pythonistes : `asyncio`.

Méconnue et encore un peu mal comprise, cette *bibliothèque standard dans la bibliothèque standard* est une véritable petite révolution dans le monde de Python, en introduisant dans le cœur du langage de nouveaux éléments syntaxiques, adaptés à un paradigme de programmation dont d'autres langages (Node.js , Go …) ou frameworks (twisted , tornado , gevent …) avaient dessiné les contours avant elle : la *programmation asynchrone.*

Mais enfin, à quoi ça sert ? Et comment ça marche ?

C'est le thème qui revient le plus quand je discute d'`asyncio` avec d'autres développeurs. On a beau sentir que cette bibliothèque a des enjeux tellement importants que le créateur de Python a travaillé dessus pendant plus d'une année entière (sous le nom de code *Tulip*), il reste difficile de comprendre ce qui légitime autant d'efforts pour inclure la programmation asynchrone dans le cœur de Python : après tout, «*c'est juste adapté au réseau et au web*», non ?

Il y a un an, je vous parlais de coroutines sur ce site, en vous expliquant que c'était la base d'`asyncio`. Aujourd'hui, mon but est à la fois de vous faire comprendre à quel besoin répond cette bibliothèque, et surtout **comment c'est fichu à l'intérieur**.

Vous vous apercevrez certainement qu'à la fin de cet article, nous n'aurons pas fait une seule ligne de programmation réseau.

C'est voulu, parce que le but est justement de vous montrer **le mécanisme**, sans le réduire à l'une de ses applications possibles.

1. Ça veut dire quoi, asynchrone ?

En un mot comme en cent, un programme qui fonctionne de façon *asychrone*, c'est un programme qui évite au maximum de passer du temps à *attendre sans rien faire*, et qui s'arrange pour *s'occuper autant que possible pendant qu'il attend*. Cette façon d'optimiser le temps d'attente est tout à fait naturelle pour nous. Par exemple, on peut s'en rendre compte en observant le travail d'un serveur qui monte votre commande dans un fast-food.

De façon synchrone :

- Préparer le hamburger:
  - Demander le hamburger en cuisine.
  - Attendre le hamburger (1 minute).
  - Récupérer le hamburger et le poser sur le plateau.
- Préparer les frites:
  - Mettre des frites à chauffer.
  - Attendre que les frites soient cuites (2 minutes).
  - Récupérer des frites et les poser sur le plateau.
- Préparer la boisson:
  - Placer un gobelet dans la machine à soda.
  - Remplir le gobelet (30 secondes).
  - Récupérer le gobelet et le poser sur le plateau.

En gros, si notre employé de fast-food était synchrone, il mettrait 3 minutes et 30 secondes pour monter votre commande. Et je vous garantis que s'il fonctionnait vraiment de cette façon, **vous ne remettriez plus jamais les pieds dans ce fast-food** ! 🍊

Schématisons ce fonctionnement. Nous avons 4 acteurs, ou "services" :

1. Le comptoir (où se trouve le serveur qui monte votre commande),
2. La cuisine,
3. La friteuse,
4. La fontaine à soda.

Voici comment les choses se dérouleraient de façon synchrone :

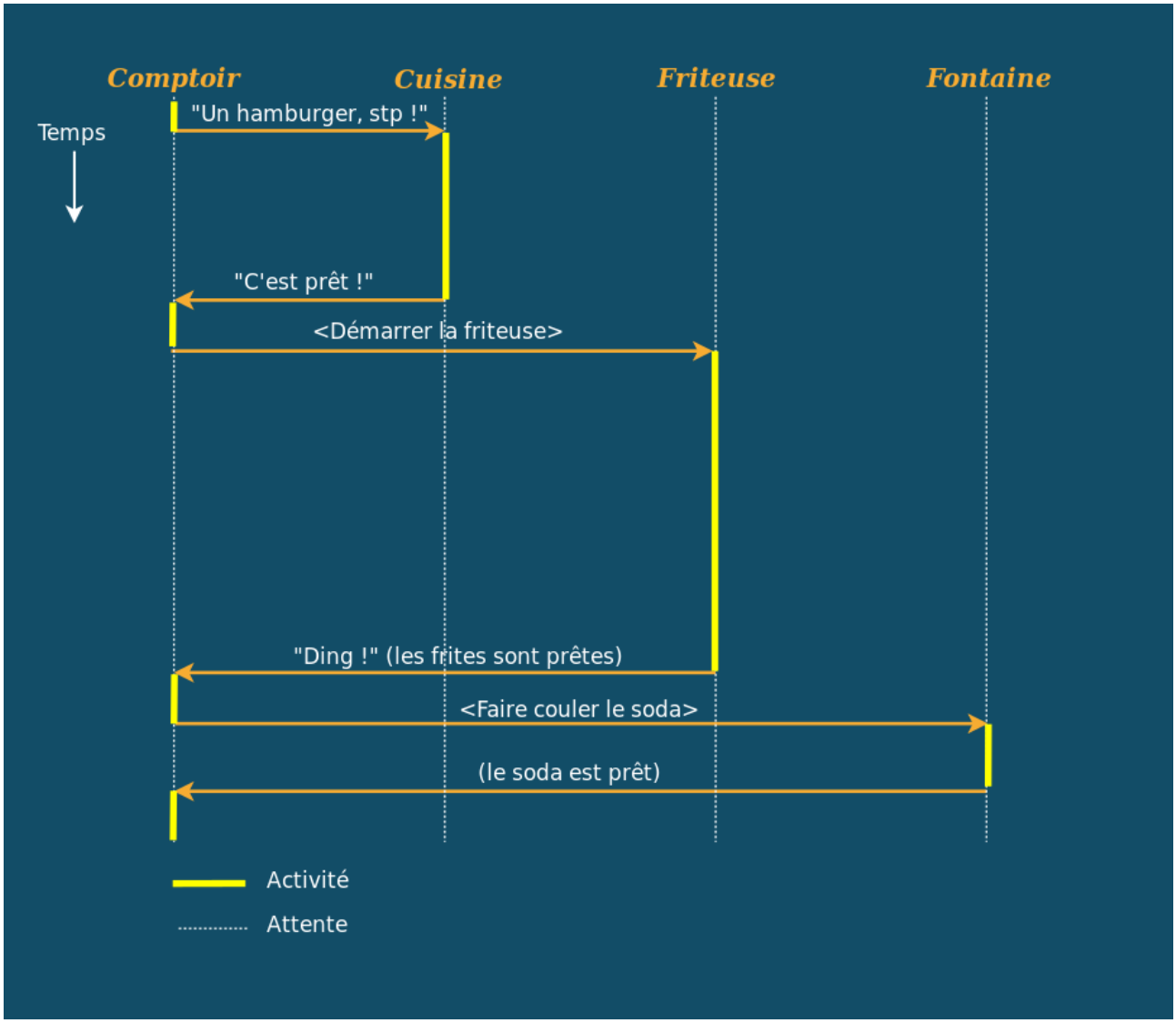

FIGURE 1.1. – Le travail d'un serveur de fast food… s'il était synchrone

Regardez la ligne du comptoir, chaque portion en pointillés signifie que le serveur reste bloqué à attendre que ça soit prêt. En informatique, on appelle cela des **entrées-sorties bloquantes**. Imaginez un peu un serveur de fast-food qui attend bêtement devant la cuisine que le hamburger arrive, avant de passer à la suite… Il aurait l'air un peu idiot, non ? Et surtout, vous seriez servi froid !

Dans la réalité, un serveur fonctionne plutôt de façon **asynchrone** :

— Demander le hamburger en cuisine.
— Mettre les frites à chauffer.
— Placer un gobelet dans la machine à soda et le mettre à remplir.
— Après 30 secondes : Récupérer le gobelet et le poser sur le plateau.
— Après 1 minute: Récupérer le hamburger et le poser sur le plateau.
— Après 2 minutes: Récupérer les frites et les poser sur le plateau.

Ce qui donne le schéma suivant :

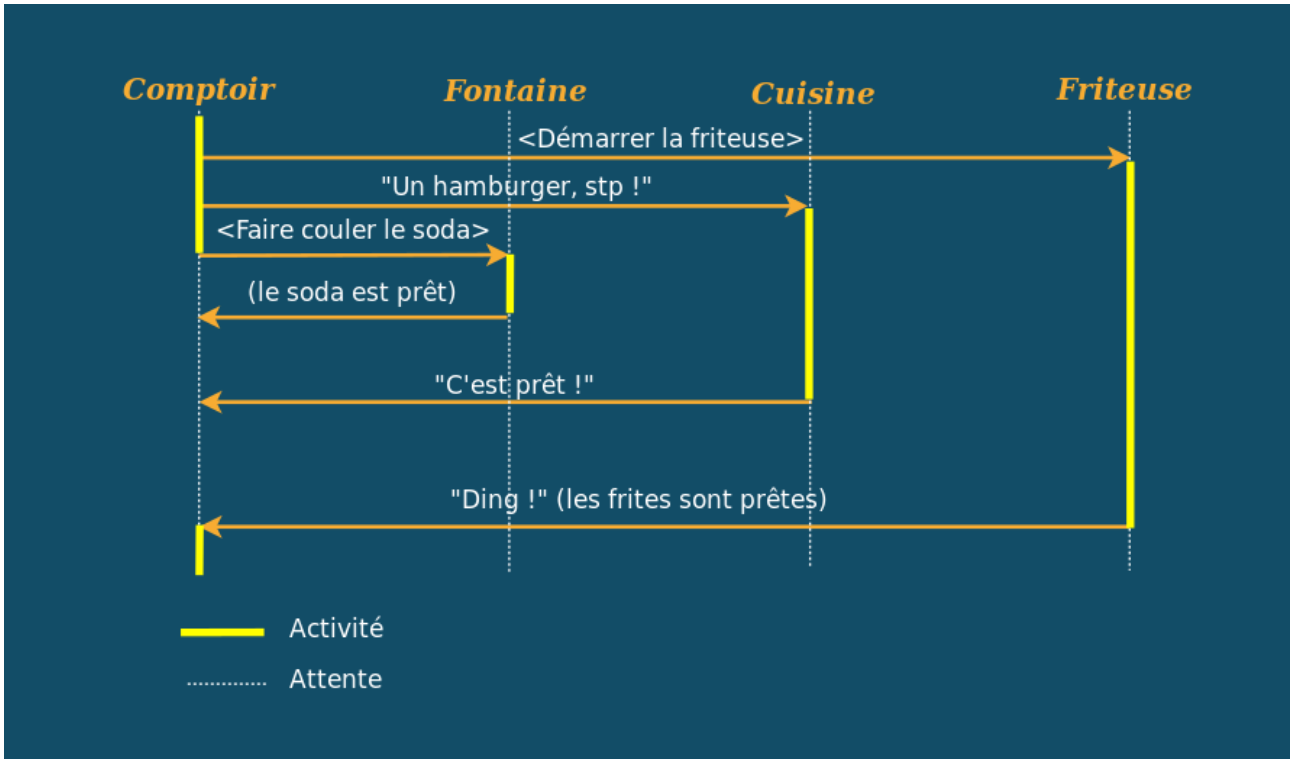

FIGURE 1.2. – Le travail d'un serveur de fast food, de façon plus réaliste

En travaillant de façon asynchrone, notre employé de fast-food monte maintenant votre commande en 2 minutes. Mais ça ne s'arrête pas là ! Regardez, bien, *il reste des pointillés* sur la ligne du comptoir. À votre avis, que fait notre serveur pendant ce temps ?

Eh bien, **il sert d'autres clients**, pardi !

— **Une commande `A` est confiée à l'employé**
— Demander le burger pour `A` en cuisine
— Mettre les frites à chauffer.
— Placer un gobelet dans la machine à soda pour `A`.
— Après 30 secondes: Récupérer le gobelet de `A` et le poser sur son plateau
— **Une nouvelle commande `B` est prise et confiée à l'employé**
— Demander le burger pour `B` en cuisine
— Placer un gobelet dans la machine à soda pour `B`.
— Après 1 minute: Le burger de `A` est prêt, le poser sur son plateau.
— La boisson de `B` est remplie, la poser sur son plateau.
— Après 1 minute 40: Le burger de `B` est prêt, le poser sur son plateau.
— Après 2 minutes: Les frites sont prêtes, servir `A` et `B`

Toujours en 2 minutes, l'employé asynchrone vient cette fois de servir 2 clients. Si vous vous mettez à la place du client `B` qui aurait dû attendre que l'employé finisse de monter la commande de `A` avant de s'occuper de la sienne dans un schéma synchrone, celui-ci a été servi en 1 minute 30 au lieu d'attendre 6 minutes 30. Voilà pourquoi on parle de **fast**-food !

Si l'on devait résumer et légitimer la programmation asynchrone en une phrase, voici la conclusion que l'on devrait tirer de cet exemple :

La programmation asynchrone est une façon de concevoir des programmes qui s'exécutent de façon **concurrente**.

Pensez-y la prochaine fois que vous irez manger dans un fast-food, et observez les serveurs. Leur boulot vous semblera d'un coup beaucoup plus compliqué qu'il n'y paraît.

2. Concurrence et parallélisme

2.1. Deux notions à ne pas confondre !

Lorsque l'on parle de **concurrence**, beaucoup pensent à l'exécution **parallèle** de plusieurs tâches. Dissipons cet amalgame au plus vite, sans quoi vous risquez de vous perdre dans la suite.

— Un programme qui s'exécute de façon **concurrente**, c'est un programme qui, à un instant T, est en train de réaliser **plusieurs tâches en même temps**, comme notre employé de fast-food qui est capable de monter plusieurs commandes à la fois.
— Un programme qui s'exécute de façon **parallèle**, c'est **UNE** tâche qui a été découpée en plusieurs morceaux pour être réalisée par **PLUSIEURS** acteurs en même temps, la plupart du temps pour qu'elle se termine plus vite.

Si vous préférez une image plus visuelle :

— Un serveur de fast-food n'a pas besoin de se dupliquer pour monter les commandes de deux clients à la fois, et surtout, même s'il en était capable, **il ne servirait pas forcément les gens plus vite**. Il exécute donc des tâches concurrentes, sans parallélisation.
— Si vous preniez le serveur ~~un peu débile~~ synchrone du premier exemple et que vous en mettiez 5 derrière un comptoir, cela vous donnerait un parfait exemple de parallélisme (mais ce serait quand même du gaspillage de ressources).

2.2. Les threads et le GIL

L'immense majorité des systèmes d'exploitation modernes propose un mécanisme natif et relativement commode pour réaliser des programmes concurrents : les threads. Un processus (au sens système) peut partager son travail en plusieurs fils d'exécution concurrents, que l'on appelle des *threads*. Ces threads ont le double avantage d'être plus légers à créer qu'un processus, et de **partager** leur mémoire, ce qui leur permet de communiquer de façon extrêmement efficace.

En fait, dans la famille des langages dits *système* (C, C++, Go, Rust…), plusieurs threads peuvent même s'exécuter en parallèle, en utilisant tous les cœurs de traitement que le système d'exploitation met à leur disposition. **Mais pas en Python**, ni dans aucun autre langage de la même famille que lui (Ruby, PHP, Javascript…). En effet, l'interpréteur Python implémente ce que l'on appelle un GIL. Ce système a pour avantage de simplifier son architecture et sa conception : un code en Python ne peut être exécuté par l'interpréteur que si celui-ci est possesseur du GIL, et bien évidemment un seul thread peut posséder le GIL à un instant donné. Ainsi, même si deux threads d'un même programme en Python sont exécutés sur deux cœurs

de processeur distincts, le GIL les contraint à ne jamais pouvoir s'exécuter en parallèle (ce qui apporte de nombreuses garanties dans le code interne de l'interpréteur).

Malgré cette contrainte, lorsqu'un thread doit réaliser une opération d'entrée-sortie (une **IO**, comme lire dans un fichier ou établir une connexion réseau), le système d'exploitation est suffisamment intelligent pour ne pas lui rendre la main tant que l'opération n'est pas terminée, ce qui fait que les threads sont une façon relativement commode de concevoir des programmes qui réalisent beaucoup d'opérations d'entrée-sortie en concurrence.

? Alors pourquoi tu nous bassines avec `asyncio` puisqu'on peut utiliser des threads ?

Je peux vous donner deux raisons principales :

1. **Le GIL coûte cher.**
2. **On ne sait jamais quand un thread va se mettre en pause.**

La première raison est bassement technique : le GIL est une force de frottement dans l'interpréteur Python. Sa gestion picore sur le temps d'exécution des threads, de façon proportionnelle au nombre de tâches concurrentes en cours d'exécution. En somme, plus il y a de threads, plus le programme est ralenti, ce qui est embarrassant dans de nombreuses applications.

Imaginez un serveur de chat réalisé avec le module `socketserver`. Chaque fois qu'un client se connecte, le serveur va lancer un nouveau thread pour gérer la connexion. Plus il y aura de gens connectés, plus le serveur sera ralenti, non pas à cause du plus grand nombre d'entrées-sorties, mais bêtement *à cause du GIL* qui va utiliser du temps de calcul uniquement pour orchestrer le travail du serveur. En somme, il arrivera un seuil au-delà duquel le serveur ne passera plus à l'échelle à cause de l'interpréteur Python et d'une contrainte sur laquelle le développeur n'a aucune maîtrise.

La seconde raison est également très importante : par nature, les threads *partagent leur mémoire*. En particulier cela leur permet d'agir sur des données partagées, de façon concurrente, sauf que vous ne pouvez jamais savoir quand un thread sera mis en pause pour laisser travailler les autres, donc si vous n'utilisez pas de mécanismes de *synchronisation* (comme des verrous, des mutexes, des sémaphores), **vous n'avez absolument aucune garantie** que personne ne viendra vous marcher sur les pieds pendant que vous touchez à une donnée. Cela rend la programmation multithread *fastidieuse*, non seulement parce que les bugs qu'elle introduit (*race conditions*, *deadlocks*) sont extrêmement difficiles à prévoir, et encore plus à diagnostiquer, mais également parce que le remède à cette catégorie de bugs a lui-même un coût (du même ordre que celui du GIL).

En somme, même s'il est très facile de multi-threader un programme sans pratiquement en modifier la source, le système de concurrence lui-même pose quelques freins à la réalisation d'applications qui passent à l'échelle et croissent en complexité avec le temps, au fil des nouvelles fonctionnalités.

2.3. Les IO asynchrones

Pour les raisons expliquées plus haut, `asyncio` propose un modèle de concurrence :

— Où les interruptions sont prédictibles et explicites, donc tout le code entre deux interruptions est *atomique* : si vous ne placez pas d'interruption explicite entre deux lignes de code celles-ci seront exécutées d'un bloc, sans risque de modification extérieure.
— Où l'ordonnancement entre les tâches n'est pas réalisé par le système d'exploitation, mais dans le *userland* et de façon plus intelligente et adaptée à la nature des tâches à exécuter en concurrence.

Vous aurez compris que les tâches concurrentes en question sont des **entrées-sorties**, ou **IO**, mais de quoi parlons-nous exactement ?

Une **IO**, au sens large, est une tâche pendant laquelle votre programme attend un résultat qui vient de l'extérieur. Dans le contexte d'un programme en Python cela peut être :

— L'échange de données sur une connexion réseau,
— L'ouverture, la lecture ou l'écriture d'un fichier,
— L'exécution d'un programme dans un sous-processus,
— L'attente d'événements qui viendraient de périphériques de la machine…

En somme, si l'on considère toutes ces opérations comme des interactions avec des services extérieurs à votre programme, celui-ci devient une sorte de grand chef d'orchestre qui se contente de communiquer avec ces services. C'est typiquement dans ce genre d'activité que la programmation asynchrone excelle. Il vous suffit de formuler votre programme en identifiant clairement quelles **IO** celui-ci doit réaliser, lesquelles peuvent s'exécuter en parallèle, et parfois même transformer certaines tâches calculatoires en IO, pour que votre programme sache non seulement réaliser chaque tâche plus rapidement qu'avant, mais qu'il devienne également capable de traiter plusieurs de ces tâches en même temps !

Je pense que vous aurez maintenant compris pourquoi les services web adoptent de plus en plus ce paradigme. Mais finissons-en avec cette trop longue introduction et commençons à regarder sous le capot, si vous le voulez bien.

3. Une boucle événementielle, c'est essentiel

La notion fondamentale autour de laquelle `asyncio` a été construite est celle de *coroutine*.

Une coroutine est une tâche qui peut décider de se suspendre elle-même au moyen du mot-clé `yield`, et attendre jusqu'à ce que le code qui la contrôle décide de lui rendre la main en *itérant* dessus.

On peut imaginer, par exemple, écrire la fonction suivante:

```
1 def tic_tac():
2     print("Tic")
3     yield
4     print("Tac")
```

```
    yield
    return "Boum!"
```

Cette fonction, puisqu'elle utilise le mot-clé `yield`, définit une *coroutine*[1]. Si on l'invoque, la fonction `tic_tac` retourne une tâche prête à être exécutée, mais n'exécute pas les instructions qu'elle contient.

```
>>> task = tic_tac()
>>> task
<generator object tic_tac at 0x7fe157023280>
```

En termes de vocabulaire, on dira que notre fonction `tic_tac` est une *fonction coroutine*, c'est-à-dire une fonction qui **construit une coroutine**. La coroutine est contenue ici dans la variable `task`.

Nous pouvons maintenant exécuter son code jusqu'au prochain `yield`, en nous servant de la fonction standard `next()` :

```
>>> next(task)
Tic
>>> next(task)
Tac
>>> next(task)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration: Boum!
```

Lorsque la tâche est terminée, une exception `StopIteration` est levée. Celle-ci contient la valeur de retour de la coroutine. Jusqu'ici, rien de bien sorcier. Dès lors, on peut imaginer créer une petite boucle pour exécuter cette coroutine jusqu'à épuisement :

```
>>> task = tic_tac()
>>> while True:
...     try:
...         next(task)
...     except StopIteration as stop:
...         print("valeur de retour:", repr(stop.value))
...         break
...
Tic
Tac
```

1. En toute rigueur il s'agit d'un *générateur*, mais comme nous avons pu l'observer dans un précédent article , les générateurs de Python sont implémentés comme de véritables coroutines.

```
valeur de retour: 'Boum!'
```

Afin de nous affranchir de la sémantique des itérateurs de Python, créons une classe `Task` qui nous permettra de manipuler nos coroutines plus aisément :

```python
STATUS_NEW = 'NEW'
STATUS_RUNNING = 'RUNNING'
STATUS_FINISHED = 'FINISHED'
STATUS_ERROR = 'ERROR'

class Task:
    def __init__(self, coro):
        self.coro = coro  # Coroutine à exécuter
        self.name = coro.__name__
        self.status = STATUS_NEW  # Statut de la tâche
        self.return_value = None  # Valeur de retour de la coroutine
        self.error_value = None  # Exception levée par la coroutine

    # Exécute la tâche jusqu'à la prochaine pause
    def run(self):
        try:
            # On passe la tâche à l'état RUNNING et on l'exécute jusqu'à
            # la prochaine suspension de la coroutine.
            self.status = STATUS_RUNNING
            next(self.coro)
        except StopIteration as err:
            # Si la coroutine se termine, la tâche passe à l'état FINISHED
            # et on récupère sa valeur de retour.
            self.status = STATUS_FINISHED
            self.return_value = err.value
        except Exception as err:
            # Si une autre exception est levée durant l'exécution de la
            # coroutine, la tâche passe à l'état ERROR, et on récupère
            # l'exception pour laisser l'utilisateur la traiter.
            self.status = STATUS_ERROR
            self.error_value = err

    def is_done(self):
        return self.status in {STATUS_FINISHED, STATUS_ERROR}

    def __repr__(self):
        result = ''
        if self.is_done():
```

```
39              result = " ({!r})".format(self.return_value or
                    self.error_value)
40
41          return "<Task '{}' [{}]{}>".format(self.name, self.status,
                result)
```

Son fonctionnement est plutôt simple. Réimplémentons notre boucle en nous servant de cette classe :

```
1  >>> task = Task(tic_tac())
2  >>> task
3  <Task 'tic_tac' [NEW]>
4  >>> while not task.is_done():
5  ...     task.run()
6  ...     print(task)
7  ...
8  Tic
9  <Task 'tic_tac' [RUNNING]>
10 Tac
11 <Task 'tic_tac' [RUNNING]>
12 <Task 'tic_tac' [FINISHED] ('Boom!')>
13 >>> task.return_value
14 'Boom!'
```

Bien. Nous avons une classe qui nous permet de manipuler des tâches en cours d'exécution, ces tâches étant implémentées sous la forme de coroutines. Il ne nous reste plus qu'à trouver un moyen d'exécuter plusieurs coroutines de façon **concurrente**.

En effet, tout l'intérêt de la programmation asynchrone est d'être capable d'occuper le programme pendant qu'une tâche donnée est en attente d'un événement, donc il faut trouver un moyen pour que, du moment qu'une tâche a décidé de se suspendre, les autres puissent se réveiller et travailler à leur tour.

Pour cela, il suffit de construire une *file d'attente* de tâches à exécuter. En Python, l'objet le plus pratique pour modéliser une file d'attente est la classe standard `collections.deque` (*double-ended queue*). Cette classe possède les mêmes méthodes que les listes, auxquelles viennent s'ajouter :

— `appendleft()` pour ajouter un élément au tout début de la liste,
— `popleft()` pour retirer (et retourner) le premier élément de la liste.

Ainsi, il suffit ajouter les éléments à une extrémité de la file (`append()`), et consommer ceux de l'autre extrémité (`popleft()`). On pourrait arguer qu'il est possible d'ajouter des éléments n'importe où dans une liste avec la méthode `insert()`, mais la classe `deque` est vraiment *faite pour* créer des files et des piles : ses opérations aux extrémités sont bien plus efficaces que la méthode `insert()`.

Essayons d'exécuter en concurrence deux instances de notre coroutine `tic_tac` :

```
>>> from collections import deque
>>> running_tasks = deque()
>>> running_tasks.append(Task(tic_tac()))
>>> running_tasks.append(Task(tic_tac()))
>>> while running_tasks:
...     # On récupère une tâche en attente et on l'exécute
...     task = running_tasks.popleft()
...     task.run()
...     if task.is_done():
...         # Si la tâche est terminée, on l'affiche
...         print(task)
...     else:
...         # La tâche n'est pas finie, on la replace au bout
...         # de la file d'attente
...         running_tasks.append(task)
...
Tic
Tic
Tac
Tac
<Task 'tic_tac' [FINISHED] ('Boom!')>
<Task 'tic_tac' [FINISHED] ('Boom!')>
```

Voilà qui est intéressant : la sortie des deux coroutines est entremêlée! Cela signifie que les deux tâches ont été exécutées simultanément, de façon **concurrente**.

Nous avons tout ce qu'il nous faut pour modéliser une boucle événementielle, c'est-à-dire une boucle qui s'occupe de programmer l'exécution et le réveil des tâches dont elle a la charge. Implémentons celle-ci dans la classe `Loop` suivante :

```
from collections import deque

class Loop:
    def __init__(self):
        self._running = deque()

    def _loop(self):
        task = self._running.popleft()
        task.run()
        if task.is_done():
            print(task)
            return
        self.schedule(task)

    def run_until_empty(self):
        while self._running:
            self._loop()
```

```
18
19      def schedule(self, task):
20          if not isinstance(task, Task):
21              task = Task(task)
22          self._running.append(task)
23          return task
```

Vérifions :

```
1  >>> def spam():
2  ...     print("Spam")
3  ...     yield
4  ...     print("Eggs")
5  ...     yield
6  ...     print("Bacon")
7  ...     yield
8  ...     return "SPAM!"
9  ...
10 >>> event_loop = Loop()
11 >>> event_loop.schedule(tic_tac())
12 >>> event_loop.schedule(spam())
13 >>> event_loop.run_until_empty()
14 Tic
15 Spam
16 Tac
17 Eggs
18 <Task 'tic_tac' [FINISHED] ('Boom!')>
19 Bacon
20 <Task 'spam' [FINISHED] ('SPAM!')>
```

Tout fonctionne parfaitement. Dotons tout de même notre classe `Loop` d'une dernière méthode pour exécuter la boucle jusqu'à épuisement d'une coroutine en particulier :

```
1  class Loop:
2      # ...
3      def run_until_complete(self, task):
4          task = self.schedule(task)
5          while not task.is_done():
6              self._loop()
```

Testons-la :

```
1  >>> event_loop = Loop()
2  >>> event_loop.run_until_complete(tic_tac())
```

```
Tic
Tac
<Task 'tic_tac' [FINISHED] ('Boom!')>
```

Pas de surprise.

Toute la programmation asynchrone repose sur ce genre de boucle qui sert en fait d'*ordonnanceur* aux tâches en cours d'exécution. Pour vous en convaincre, regardez ce bout de code qui utilise `asyncio` :

```
>>> import asyncio
>>> loop = asyncio.get_event_loop()
>>> loop.run_until_complete(tic_tac())
Tic
Tac
'Boom!'
>>> loop.run_until_complete(asyncio.wait([tic_tac(), spam()]))
Spam
Tic
Eggs
Tac
Bacon
({Task(<tic_tac>)<result='Boom!'>, Task(<spam>)<result='SPAM!'>},
    set())
```

Drôlement familier, n'est-ce pas ? Ne bloquez pas sur la fonction `asyncio.wait`: il s'agit simplement d'une coroutine qui sert à lancer plusieurs tâches en même temps et attendre que celles-ci se terminent avant de retourner.

Les `yield` sont des "interruptions système" !

Dans la réalité, la boucle événementielle réalise un peu plus de travail que ce que nous venons de faire. En particulier, elle est capable de dire, à chaque instant, quelles tâches sont en attente de quelle IO, et de les réveiller lorsque l'IO en question est terminée. Nous ne coderons pas ce mécanisme dans cet article (car il ajouterait pas mal de complexité pour peu de choses), mais il est important que vous compreniez dès maintenant que **chaque fois qu'une coroutine se met en attente d'une IO ou d'un autre événement, celle-ci se suspend avec `yield`**.

Si l'on fait un parallèle avec la programmation système, on peut considérer un `yield` comme une *interruption système* pendant laquelle un processus laisse la main au noyau du système d'exploitation. C'est grâce à ce mécanisme que sont réalisés les appels système, y compris ceux qui servent à réaliser des IO.

Cela veut dire que dans un code asynchrone "de la vraie vie" on n'écrit **jamais** explicitement `yield` ; on appelle plutôt des coroutines "natives" qui le font pour nous. Ces coroutines spéciales peuvent êtres vues comme des appels-système ("ouvre ce fichier", "attends qu'il y ait quelque chose à lire sur cette socket", "réveille moi dans 3 secondes"…).

4. Appels de coroutines

Vous aurez donc compris qu'un programme asynchrone doit être écrit dans des *coroutines* puisque tout repose sur le fait que celles-ci sont interruptibles.

Supposons maintenant qu'une coroutine ait besoin de faire appel à une autre coroutine, pour lui déléguer du travail ou bien lui demander comme nous l'avons mentionné juste au-dessus, de réaliser une IO. Nous avons alors deux cas de figure :

— Soit nous voulons appeler cette nouvelle coroutine de façon *séquentielle*, et lui laisser la main en attendant qu'elle ait fini de travailler.
— Soit nous voulons que cette nouvelle coroutine s'exécute de façon *concurrente*.

Voyons un peu comment cela se passe.

4.1. Appel séquentiel

Pour le premier cas de figure, rappelons d'abord que la syntaxe `yield from` introduite dans le langage depuis Python 3.3 nous permet de passer la main à une autre coroutine. Par exemple, dans le code suivant, la coroutine `example` utilise cette syntaxe pour laisser temporairement la main à la coroutine `subtask` :

```
1  def example():
2      print("Tâche 'example'")
3      print("Lancement de la tâche 'subtask'")
4      yield from subtask()
5      print("Retour dans 'example'")
6      for _ in range(3):
7          print("(example)")
8          yield
9
10 def subtask():
11     print("Tâche 'subtask'")
12     for _ in range(2):
13         print("(subtask)")
14         yield
```

Vérifions :

```
1  >>> event_loop = Loop()
2  >>> event_loop.run_until_complete(example())
3  Tâche 'example'
4  Lancement de la tâche 'subtask'
5  Tâche 'subtask'
6  (subtask)
7  (subtask)
```

```
Retour dans 'example'
(example)
(example)
(example)
<Task 'example' [FINISHED] (None)>
```

Ainsi, Python nous fournit déjà nativement un élément de syntaxe pour *lancer une tâche de façon séquentielle* à l'intérieur d'une coroutine : `yield from`, tout simplement.

4.2. Lancement d'une tâche concurrente

Pour lancer une tâche concurrente, il suffit de la programmer dans la boucle événementielle. `asyncio` nous propose pour cela une fonction `ensure_future()` qui permet de le faire avec sa boucle événementielle par défaut. Voici comment nous pourrions la reproduire dans notre mini-framework :

```
DEFAULT_LOOP = Loop()


def ensure_future(coro, loop=None):
    if loop is None:
        loop = DEFAULT_LOOP
    return loop.schedule(coro)
```

Modifions un notre coroutine `example` en conséquence :

```
def example():
    print("Tâche 'example'")
    print("Lancement de la tâche 'subtask'")
    ensure_future(subtask())   # <- appel à ensure_future au lieu
        de yield from
    print("Retour dans 'example'")
    for _ in range(3):
        print("(example)")
        yield
```

Et voilà le résultat :

```
>>> event_loop = DEFAULT_LOOP
>>> event_loop.run_until_complete(example())
Tâche 'example'
Lancement de la tâche 'subtask'
Retour dans 'example'
```

```
(example)
Tâche 'subtask'
(subtask)
(example)
(subtask)
(example)
<Task 'subtask' [FINISHED] (None)>
<Task 'example' [FINISHED] (None)>
```

Magique, n'est-ce pas ?

Par contre, une fois que notre coroutine est lancée, nous n'avons pas tout à fait le contrôle de son exécution. Par exemple, si nous rendions la tâche `subtask` plus longue qu'`example`, celle-ci lui «survivrait» :

```
>>> def subtask():
...     print("Tâche 'subtask'")
...     for _ in range(5):
...         print("(subtask)")
...         yield
...
>>> event_loop.run_until_complete(example())
Tâche 'example'
Lancement de la tâche 'subtask'
Retour dans 'example'
(example)
Tâche 'subtask'
(subtask)
(example)
(subtask)
(example)
(subtask)
<Task 'example' [FINISHED] (None)>
```

L'exécution s'arrête avec la fin de la coroutine `example`, mais la coroutine `subtask`, elle, n'a pas fini. Elle est encore suspendue dans la boucle, à l'état de zombie alors que le reste du programme est terminé. Vidons ce qu'il reste dans la boucle événementielle :

```
>>> event_loop.run_until_empty()
(subtask)
(subtask)
<Task 'subtask' [FINISHED] (None)>
```

4.3. Annulation d'une tâche

Que faire si nous ne voulons pas qu'une coroutine quitte avant une sous-tâche qu'elle aurait lancée en parallèle ?

Nous avons deux solutions. La première, dont nous nous contenterons dans cet exemple, serait de pouvoir *annuler* une tâche en cours d'exécution. Il nous suffit pour cela de créer un nouvel état dans notre classe `Task` :

```
STATUS_CANCELLED = "CANCELLED"

class Task:

    # ...

    def cancel(self):
        if self.is_done():
            # Inutile d'annuler une tâche déjà terminée
            return
        self.status = STATUS_CANCELLED

    def is_cancelled(self):
        return self.status == STATUS_CANCELLED
```

Rajoutons un test dans la boucle événementielle pour déprogrammer les tâches annulées :

```
class Loop:

    # ...

    def _loop(self):
        task = self._running.popleft()

        if task.is_cancelled():
            # Si la tâche a été annulée,
            # on ne l'exécute pas et on "l'oublie".
            print(task)
            return

        # ... le reste de la méthode est identique
```

Il ne nous reste plus qu'une petite coroutine utilitaire à écrire pour annuler une tâche en cours d'exécution :

```
def cancel(task):
    # On annule la tâche
    task.cancel()
    # On laisse la main à la boucle événementielle pour qu'elle
        ait l'occasion
    # de prendre en compte l'annulation
    yield

def example():
    print("Tâche 'example'")
    print("Lancement de la tâche 'subtask'")
    sub = ensure_future(subtask())
    print("Retour dans 'example'")
    for _ in range(3):
        print("(example)")
        yield
    yield from cancel(sub)
```

Vérifions :

```
>>> event_loop.run_until_complete(example())
Tâche 'example'
Lancement de la tâche 'subtask'
Retour dans 'example'
(example)
Tâche 'subtask'
(subtask)
(example)
(subtask)
(example)
(subtask)
<Task 'subtask' [CANCELLED]>
<Task 'example' [FINISHED] (None)>
```

Notre mécanisme d'annulation fonctionne comme prévu. Cela dit, on peut aussi imaginer tout simplement vouloir *attendre* de façon asynchrone que la sous-tâche ait terminé son exécution avant de quitter proprement, et c'est la raison d'être de la coroutine `asyncio.wait()` que je vous ai montrée plus haut. 🍊

5. La syntaxe asynchrone de Python 3.5

Maintenant que nous avons compris comment la boucle d'`asyncio` se débrouille pour exécuter des coroutines de façon concurrente, il est temps de l'utiliser pour de bon.

Commençons par adopter la syntaxe de Python 3.5. En réalité, la boucle événementielle et la fonction `ensure_future()` que nous avons programmées jusqu'à présent respectent exactement la même interface que celles d'asyncio. Cela dit, même s'il est absolument possible de continuer à définir des coroutines sous la forme de générateurs, Python 3.5 a introduit la syntaxe suivante :

— Les coroutines se différencient des fonctions classiques en étant définies via la syntaxe `async def coroutine(...)` au lieu de `def coroutine(...)`.
— À l'intérieur d'une coroutine, on utilisera le mot-clé `await` au lieu de `yield from` lorsque nous voudrons appeler séquentiellement une autre coroutine.

C'est quasiment tout ce qu'il y a à savoir sur la syntaxe (hormis quelques détails que nous découvrirons un peu plus loin).

Que diriez-vous maintenant d'implémenter notre serveur de fast-food ? Cet exemple va nous servir de fil rouge jusqu'à la fin de cet article. Dans celui-ci, nous allons nous contenter de *simuler* des entrées-sorties en appelant la coroutine `asyncio.sleep`.

```
import asyncio
from datetime import datetime


async def get_soda(client):
    print("    > Remplissage du soda pour {}".format(client))
    await asyncio.sleep(1)
    print("    < Le soda de {} est prêt".format(client))

async def get_fries(client):
    print("    > Démarrage de la cuisson des frites pour {}".forma
        t(client))
    await asyncio.sleep(4)
    print("    < Les frites de {} sont prêtes".format(client))

async def get_burger(client):
    print("    > Commande du burger en cuisine pour {}".format(cli
        ent))
    await asyncio.sleep(3)
    print("    < Le burger de {} est prêt".format(client))

async def serve(client):
    print("=> Commande passée par {}".format(client))
    start_time = datetime.now()
    await asyncio.wait(
        [
            get_soda(client),
            get_fries(client),
            get_burger(client)
        ]
    )
    total = datetime.now() - start_time
```

```
31      print("<= {} servi en {}".format(client, datetime.now() -
            start_time))
32      return total
```

Rien de franchement dépaysant. Pour exécuter ce code, là aussi l'API est sensiblement la même que notre classe `Loop` :

```
1  >>> loop = asyncio.get_event_loop()
2  >>> loop.run_until_complete(serve("A"))
3  => Commande passée par A
4      > Remplissage du soda pour A
5      > Commande du burger en cuisine pour A
6      > Démarrage de la cuisson des frites pour A
7      < Le soda de A est prêt
8      < Le burger de A est prêt
9      < Les frites de A sont prêtes
10 <= A servi en 0:00:04.003105
```

Pas d'erreur de syntaxe, le code fonctionne. On peut commencer à travailler.

6. Le problème du fast-food

Remarquons dans un premier temps que notre serveur **manque de réalisme**. En effet, si nous lui demandons de servir deux clients en même temps, voilà ce qui se produit :

```
1  >>> loop.run_until_complete(
2  ...     asyncio.wait([serve("A"), serve("B")])
3  ... )
4  => Commande passée par A
5  => Commande passée par B
6      > Remplissage du soda pour A
7      > Commande du burger en cuisine pour A
8      > Démarrage de la cuisson des frites pour A
9      > Démarrage de la cuisson des frites pour B
10     > Remplissage du soda pour B
11     > Commande du burger en cuisine pour B
12     < Le soda de A est prêt
13     < Le soda de B est prêt
14     < Le burger de A est prêt
15     < Le burger de B est prêt
16     < Les frites de A sont prêtes
17     < Les frites de B sont prêtes
18 <= A servi en 0:00:04.002609
19 <= B servi en 0:00:04.002792
```

Les deux commandes ont été servies simultanément, de la même façon. La préparation des trois ingrédients s'est chevauchée, comme s'il était possible de faire couler une infinité de sodas, de cuire une infinité de frites *à la demande* pour les clients, et de préparer une infinité de hamburgers en parallèle.

En bref : **notre modélisation manque de contraintes**.

Pour améliorer ce programme, nous allons modéliser les contraintes suivantes :

— La machine à sodas ne peut faire couler **qu'un seul soda à la fois**. Dans une application réelle, cela reviendrait à *requêter un service synchrone qui ne supporte pas les accès concurrents* ;
— Il n'y a que 3 cuisiniers dans le restaurant, donc **on ne peut pas préparer plus de trois hamburgers en même temps**. Dans la réalité, cela revient à *requêter un service synchrone dont trois instances tournent en parallèle* ;
— Le bac à frites s'utilise en faisant cuire 5 portions de frites d'un coup, pour servir ensuite 5 clients instantanément. Dans la réalité, cela revient, à peu de choses près, à *simuler un service synchrone qui fonctionne avec un cache.*

La machine à soda est certainement la plus simple. Il est possible de verrouiller une ressource de manière à ce qu'une seule tâche puisse y accéder à la fois, en utilisant ce que l'on appelle un **verrou** (`asyncio.Lock`). Plaçons un verrou sur notre machine à soda :

```
1  SODA_LOCK = asyncio.Lock()
2
3  async def get_soda(client):
4      # Acquisition du verrou
5      # la syntaxe 'async with FOO' peut être lue comme 'with (yield
           from FOO)'
6      async with SODA_LOCK:
7          # Une seule tâche à la fois peut exécuter ce bloc
8          print("    > Remplissage du soda pour {}".format(client))
9          await asyncio.sleep(1)
10         print("    < Le soda de {} est prêt".format(client))
```

Le `async with SODA_LOCK` signifie que lorsque le serveur arrive à la machine à soda pour y déposer un gobelet :

— soit la machine est libre (déverrouillée), auquel cas il peut la verrouiller pour l'utiliser immédiatement,
— soit celle-ci est déjà en train de fonctionner, auquel cas il attend (de façon asynchrone, donc en rendant la main) que le soda en cours de préparation soit prêt avant de verrouiller la machine à son tour.

Passons à la cuisine. Seuls 3 burgers peuvent être fabriqués en même temps. Cela peut se modéliser en utilisant un **sémaphore** (`asyncio.Semaphore`), qui est une sorte de "verrou multiple". On l'utilise pour qu'au plus N tâches puissent exécuter un morceau de code à un instant donné.

```
BURGER_SEM = asyncio.Semaphore(3)

async def get_burger(client):
    print("    > Commande du burger en cuisine pour {}".format(cli
        ent))
    async with BURGER_SEM:
        await asyncio.sleep(3)
        print("    < Le burger de {} est prêt".format(client))
```

Le `async with BURGER_SEM` veut dire que lorsqu'une commande est passée en cuisine :

— soit il y a un cuisinier libre, et celui-ci commence immédiatement à préparer le hamburger,
— soit tous les cuisiniers sont occupés, auquel cas on attend qu'il y en ait un qui se libère pour s'occuper de notre hamburger.

Passons enfin au bac à frites. Cette fois, `asyncio` ne nous fournira pas d'objet magique, donc il va nous falloir réfléchir un peu plus. Il faut que l'on puisse l'utiliser *une fois* pour faire les frites des 5 prochaines commandes. Dans ce cas, un compteur semble une bonne idée:

— Chaque fois que l'on prend une portion de frites, on décrémente le compteur ;
— S'il n'y a plus de frites dans le bac, il faut en refaire.

Mais attention, si les frites sont déjà en cours de préparation, il est inutile de lancer une nouvelle fournée !

Voici comment on pourrait s'y prendre :

```
FRIES_COUNTER = 0
FRIES_LOCK = asyncio.Lock()

async def get_fries(client):
    global FRIES_COUNTER
    async with FRIES_LOCK:
        print("    > Récupération des frites pour {}".format(clien
            t))
        if FRIES_COUNTER == 0:
            print("   ** Démarrage de la cuisson des frites")
            await asyncio.sleep(4)
            FRIES_COUNTER = 5
            print("   ** Les frites sont cuites")
        FRIES_COUNTER -= 1
        print("    < Les frites de {} sont prêtes".format(client))
```

Dans cet exemple, on place un verrou sur le bac à frites pour qu'un seul serveur puisse y accéder à la fois. Lorsqu'un serveur arrive devant le bac à frites, soit celui-ci contient encore des portions de frites, auquel cas il en récupère une et retourne immédiatement, soit le bac est vide, donc le serveur met des frites à cuire avant de pouvoir en récupérer une portion.

À l'exécution :

```
>>> loop.run_until_complete(asyncio.wait([serve('A'), serve('B')]))
=> Commande passée par B
=> Commande passée par A
    > Remplissage du soda pour B
    > Récupération des frites pour B
   ** Démarrage de la cuisson des frites
    > Commande du burger en cuisine pour B
    > Commande du burger en cuisine pour A
    < Le soda de B est prêt
    > Remplissage du soda pour A
    < Le soda de A est prêt
    < Le burger de B est prêt
    < Le burger de A est prêt
   ** Les frites sont cuites
    < Les frites de B sont prêtes
    > Récupération des frites pour A
    < Les frites de A sont prêtes
<= B servi en 0:00:04.003111
<= A servi en 0:00:04.003093
```

Nos deux tâches prennent toujours le même temps à s'exécuter, mais s'arrangent pour ne pas accéder simultanément à la machine à sodas ni au bac à frites.

Voyons maintenant ce que cela donne si 10 clients passent commande en même temps :

```
>>> loop.run_until_complete(
...     asyncio.wait([serve(clt) for clt in 'ABCDEFGHIJ'])
... )
...
# ... sortie filtrée ...
<= C servi en 0:00:04.004512
<= D servi en 0:00:04.004378
<= E servi en 0:00:04.004262
<= F servi en 0:00:06.008072
<= A servi en 0:00:06.008074
<= G servi en 0:00:08.006399
<= H servi en 0:00:09.009187
<= B servi en 0:00:09.009118
<= I servi en 0:00:09.015023
<= J servi en 0:00:12.011539
```

On se rend compte que les performances de notre serveur de fast-food se dégradent: certains clients attendent jusqu'à trois fois plus longtemps que les autres.

Cela n'a rien de surprenant. En fait, les performances d'une application asynchrone ne se mesurent pas en *nombre de tâches traitées simultanément*, mais plutôt, comme n'importe quel serveur, en *nombre de tâches traitées dans le temps*. Il est évident que si 10 clients viennent

manger dans un fast-food, il y a relativement peu de chances qu'ils arrivent tous en même temps : ils vont plutôt passer leur commande à raison d'une par seconde, par exemple.

Par contre, il est très important de noter que c'est bien *le temps d'attente* individuel de chaque client qui compte pour mesurer les performances (la qualité) du service. Si un client attend trop longtemps, il ne sera pas satisfait, peu importe s'il est tout seul dans le restaurant ou que celui-ci est bondé.

Pour ces raisons, il faut que nous ayons une idée des **objectifs de performances** de notre serveur, c'est-à-dire que nous fixions, comme but :

— un *temps d'attente maximal* à ne pas dépasser pour servir un client,
— un *volume* de requêtes à tenir par seconde.

Écrivons maintenant une coroutine pour tester les performances de notre serveur :

```
async def perf_test(nb_requests, period, timeout):
    tasks = []
    # On lance 'nb_requests' commandes à 'period' secondes
        d'intervalle
    for idx in range(1, nb_requests + 1):
        client_name = "client_{}".format(idx)
        tsk = asyncio.ensure_future(serve(client_name))
        tasks.append(tsk)
        await asyncio.sleep(period)

    finished, _ = await asyncio.wait(tasks)
    success = set()
    for tsk in finished:
        if tsk.result().seconds < timeout:
            success.add(tsk)

    print("{}/{} clients satisfaits".format(len(success),
        len(finished)))
```

Cette coroutine va lancer un certain nombre de commandes, régulièrement, et compter à la fin le nombre de commandes qui ont été honorées dans les temps.

Essayons de lancer 10 commandes à 1 seconde d'intervalle, avec pour objectif que les clients soient servis en 5 secondes maximum :

```
>>> loop.run_until_complete(perf_test(10, 1, 5))
# ... sortie filtrée ...
<= client_1 servi en 0:00:04.004044
<= client_2 servi en 0:00:03.002792
<= client_3 servi en 0:00:03.003338
<= client_4 servi en 0:00:03.003653
<= client_5 servi en 0:00:03.003815
<= client_6 servi en 0:00:04.003746
```

```
<= client_7 servi en 0:00:03.003412
<= client_8 servi en 0:00:03.002512
<= client_9 servi en 0:00:03.003409
<= client_10 servi en 0:00:03.003622
10/10 clients satisfaits
```

Ce test nous indique que notre serveur tient facilement une charge d'un client par seconde. Essayons de monter en charge en passant à deux clients par seconde :

```
>>> loop.run_until_complete(perf_test(10, 0.5, 5))
# ... sortie filtrée ...
<= client_1 servi en 0:00:04.002629
<= client_2 servi en 0:00:03.502093
<= client_3 servi en 0:00:03.002863
<= client_4 servi en 0:00:04.500168
<= client_5 servi en 0:00:04.500226
<= client_6 servi en 0:00:05.499894
<= client_7 servi en 0:00:05.999704
<= client_8 servi en 0:00:05.998824
<= client_9 servi en 0:00:05.999883
<= client_10 servi en 0:00:07.498776
5/10 clients satisfaits
```

À deux clients par seconde, notre serveur n'offre plus de performances satisfaisantes pour la moitié des commandes.

Nous pouvons donc poser le problème d'optimisation suivant : le gérant du restaurant veut devenir capable de servir 2 clients par seconde avec un temps de traitement inférieur à 5 secondes par commande. Pour cela, il peut :

— Acheter de nouvelles machines à sodas;
— Embaucher de nouveaux cuisiniers ;
— Remplacer son bac à frites (capable de cuire 5 portions en 4 secondes) par un nouveau, qui peut faire cuire 8 portions en 4 secondes.

Évidemment, chacune de ces solutions a un coût, donc il est préférable pour le gérant de n'apporter que le moins possible de modifications pour tenir son objectif. Si l'on voulait faire une analogie avec une application réelle:

— Acheter une seconde machine à sodas coûterait une augmentation de à 100% du CPU + une augmentation de 100% de la RAM consommés par le service "soda".
— Embaucher un quatrième cuisinier coûterait 33% de CPU supplémentaire (puisqu'il n'y a actuellement que 3 cuisiniers) + une augmentation de 33% de la RAM consommée par le service "cuisine".
— Le remplacement du bac à frites augmenterait uniquement de 60% la consommation de RAM de ce service…

En guise d'exercice, vous pouvez vous amuser à modifier les contraintes de notre programme en conséquence pour observer l'impact de vos modifications sur les performances du serveur. 🍊

Vous trouverez une solution dans le bloc masqué ci-dessous.

Contenu masqué n°1

Conclusion

Ainsi s'achève votre initiation à `asyncio`. Cet article avait pour but de vous montrer :

— l'intérêt de ce style programmation,
— la façon dont tout cela est rendu possible grâce aux coroutines de Python,
— la tête d'un code qui utilise `asyncio`.

J'espère avoir atteint ces trois objectifs, et vous avoir donné envie d'explorer plus avant cette bibliothèque. Il y a tant de choses à voir et à faire dessus que je me les réserve pour d'autres articles, à commencer par vous montrer des exemples de programmes bien réels tirant parti d'`asyncio`.

Je souhaite remercier Vayel pour ses relectures attentives et ses nombreuses questions pertinentes.

Contenu masqué

Contenu masqué n°1

Pour servir une moyenne de 2 clients par seconde, le plus logique serait de se dire qu'il faut que chaque service soit capable de tenir ce rythme en régime établi. Prenons-les un par un :

— La fontaine à sodas prend 1 seconde pour réaliser 1 soda. Si on en ajoute une deuxième, on devient alors capable de faire couler 2 sodas par seconde en moyenne.
— Actuellement, on a 3 cuisiniers capables de préparer chacun un hamburger en 3 secondes : si on veut préparer 6 hamburgers en 3 secondes, il suffit d'avoir 6 cuisiniers, donc en embaucher 3 de plus.
— Le bac à frites prépare 5 portions en 4 secondes, si on augmente sa capacité de stockage pour qu'elle soit au moins égale à 8 portions, on pourrait servir 8 portions en 4 secondes, donc une moyenne d'une portion toutes les demi-secondes.

Cela dit, imaginons que l'on n'ait pas les moyens d'embaucher un sixième cuisinier, on perd alors une demi-seconde tous les 6 clients servis, donc on sait qu'on ne pourra pas tenir le rythme indéfiniment. Une question intéressante à poser serait la suivante : est-ce que l'impact de ce retard ne peut pas être limité, en période de *rush*, en investissant notre argent dans une friteuse encore un peu plus grande qui produirait 10 portions de frites toutes les 4 secondes ? Retourner au texte.

Liste des abréviations

GIL Global Interpreter Lock. 1, 5, 6