

Découvrons la programmation asynchrone en quelques exemples

Arnaud Calmettes (nohar)

26 septembre 2015

Ça veut dire quoi, *asynchrone* ?

En un mot comme en cent, un programme qui fonctionne de façon *asynchrone*, 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.

Alors que 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.

En travaillant de façon asynchrone, notre employé de *fast food* monte maintenant votre commande en 2 minutes. Mais ça ne s'arrête pas là !

- 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.

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 !

En informatique, il existe un type de tâche qui impose aux programmes d'attendre sans rien faire : ce sont les *entrées/sorties* (ou *IO*). Nous verrons dans cet article que la programmation asynchrone est une façon extrêmement puissante d'implémenter des programmes qui réalisent plus d'IO que de calcul (comme une application de messagerie instantanée, par exemple).

Exemple n°1 : 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 :

```
def tic_tac():
    print("Tic")
    yield
    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>
```

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.

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
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 :

```
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():
        result = " ({!r})".format(self.return_value or self.error_value)

    return "<Task '{} {} [{}]>".format(self.name, self.status, result)

```

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

```

>>> task = Task(tic_tac())
>>> task
<Task 'tic_tac' [NEW]>
>>> while not task.is_done():
...     task.run()
...     print(task)
...
Tic
<Task 'tic_tac' [RUNNING]>
Tac
<Task 'tic_tac' [RUNNING]>
<Task 'tic_tac' [FINISHED] ('Boom!')>
>>> task.return_value
'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**, c'est-à-dire en parallèle les unes des autres. 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.

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 parallèle 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()

def schedule(self, task):
    if not isinstance(task, Task):
        task = Task(task)
    self._running.append(task)
    return task

```

Vérifions :

```

>>> def spam():
...     print("Spam")
...     yield
...     print("Eggs")
...     yield
...     print("Bacon")
...     yield
...     return "SPAM!"
...
>>> event_loop = Loop()
>>> event_loop.schedule(tic_tac())
>>> event_loop.schedule(spam())
>>> event_loop.run_until_empty()
Tic
Spam
Tac
Eggs
<Task 'tic_tac' [FINISHED] ('Boom!')>
Bacon
<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 :

```

class Loop:
    # ...
    def run_until_complete(self, task):
        task = self.schedule(task)
        while not task.is_done():
            self._loop()

```

Testons-la :

```

>>> event_loop = Loop()
>>> 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 parallèle et attendre que celles-ci se terminent avant de retourner. Nous la reprogrammerons nous-mêmes très bientôt. ;)

Exemple n°2 : Exécuter des tâches concurrentes

Dans le précédent exemple, nous avons déclenché l'exécution concurrente de deux coroutines en les programmant explicitement dans la boucle :

```
>>> event_loop = Loop()
>>> event_loop.schedule(Task(tic_tac()))
>>> event_loop.schedule(Task(spam()))
>>> event_loop.run_until_empty()
# les deux coroutines s'exécutent en parallèle.
```

Ce petit morceau de code nous a permis de prouver qu'avec une boucle événementielle et des coroutines, on avait tout ce qu'il nous fallait pour définir un modèle d'exécution concurrent.

Cependant, dans la pratique, on ne devrait pas avoir à manipuler directement la boucle pour réaliser quelque chose d'aussi simple que le lancement d'une tâche parallèle. En fait, **nos classes `Loop` et `Task` sont l'implémentation à bas niveau d'un petit framework asynchrone**, le programmeur ne devrait jamais avoir à y toucher à part pour lancer le programme asynchrone qu'il a écrit dans des coroutines.

Si l'on se réfère aux autres modes de concurrence :

- On peut lancer un *thread* et attendre la fin de l'exécution de celui-ci depuis n'importe quel *thread* en cours d'exécution.

- On peut créer, attendre ou arrêter un processus depuis n'importe quel processus en cours d'exécution.

Qu'il s'agisse de *threads* ou de processus, ces actions ne requièrent à aucun moment d'interagir directement avec l'ordonnanceur de votre système d'exploitation (OS).²

De façon analogue, **on devrait pouvoir lancer, attendre la fin de l'exécution, ou annuler une coroutine depuis n'importe quelle coroutine en cours d'exécution**, sans avoir à toucher directement à la boucle événementielle.

Commençons par chercher le moyen de faire appel à une coroutine depuis une tâche en cours d'exécution. Rappelons d'abord que la syntaxe `yield from` introduite dans le langage depuis Python 3.3 nous permet déjà 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` :

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

def subtask():
    print("Tâche 'subtask'")
    for _ in range(2):
        print("(subtask)")
        yield
```

Vérifions :

```
>>> event_loop = Loop()
>>> event_loop.run_until_complete(example())
Tâche 'example'
Lancement de la tâche 'subtask'
Tâche 'subtask'
(subtask)
(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. Mais ce n'est pas tout à fait ce que nous voulons : notre but, ici, est de réussir à lancer la coroutine `subtask` de façon qu'elle s'exécute *en parallèle* de la coroutine `example`, et non d'attendre que `subtask` soit terminée pour reprendre l'exécution de `example`.

2. D'ailleurs, le noyau de l'OS est justement là pour vous empêcher de toucher vous-même à l'ordonnanceur !

Pour ce faire, nous allons tirer parti d'une particularité des coroutines que nous avons découverte [dans cet article](#) : il est possible d'échanger des messages avec une coroutine en cours d'exécution.

Pour bien comprendre ce qui va suivre, continuons, si vous le voulez bien, notre parallèle avec le fonctionnement d'un système d'exploitation. Lorsqu'un processus³ A a besoin d'exécuter un programme B dans un processus concurrent, celui-ci réalise ce que l'on appelle un *appel système*, c'est-à-dire qu'il *envoie un message* à l'OS pour lui demander de bien vouloir lancer le nouveau programme B.⁴ Dans la pratique, cet *appel système* ressemble à s'y méprendre à n'importe quel appel de fonction. **L'idée-clé, c'est que le processus peut *communiquer* avec le noyau du système d'exploitation en échangeant des *messages* avec lui.**

Nous allons maintenant transposer cette notion à notre cas : nous voulons réussir à faire communiquer la boucle événementielle avec les coroutines qu'elle exécute grâce à des *messages*.

Rappelons rapidement que l'on peut envoyer une donnée à une coroutine au moment où celle-ci se suspend, en utilisant la méthode `send` des coroutines :

```
>>> def receiver():
...     while True:
...         data = yield
...         print('received:', repr(data))
...
>>> coro = receiver()
>>> next(coro) # La coroutine doit être démarrée pour recevoir des données
>>> coro.send("spam")
received: 'spam'
>>> coro.send("eggs")
received: 'eggs'
>>> coro.send("bacon")
received: 'bacon'
```

À l'opposé, on peut également envoyer des données depuis une coroutine en passant un argument au mot-clé `yield` :

```
>>> def sender():
...     while True:
...         yield "spam"
...         yield "eggs"
...         yield "bacon"
...
>>> coro = sender()
>>> coro.send(None) # Équivalent à next(coro)
'spam'
>>> coro.send(None)
'eggs'
>>> coro.send(None)
'bacon'
```

On peut donc parfaitement imaginer simuler un appel système :

3. Un *processus* peut être vu comme un **programme en cours d'exécution**.

4. Ce comportement n'est pas exclusif aux processus : de la même façon, un *thread* A envoie au noyau un appel système lorsqu'il veut lancer un nouveau *thread* B.

```
>>> def requester():
...     data = yield 'REQUÊTE'
...     print('data:', repr(data))
...     return
...
>>> coro = requester()
>>> coro.send(None)  # On lance la coroutine
'REQUÊTE'
```

La coroutine vient de nous envoyer un message. Répondons-lui.

```
>>> coro.send('RÉPONSE')
data: 'RÉPONSE'
```

Comme vous le constatez, la syntaxe de Python rend assez intuitif le fait que notre coroutine peut envoyer des requêtes à son environnement d'exécution, et se suspendre jusqu'à ce qu'on lui envoie le résultat de cette requête.

Pour que ces messages soient transmis par notre classe `Task` du premier exemple, nous devons l'accomoder un petit peu :

```
class Task:
    def __init__(self):
        # ...
        self.msg = None

    def run(self):
        try:
            self.status = STATUS_RUNNING
            return self.coro.send(self.msg)
        # ...
```

Ainsi, on peut envoyer un message à une tâche simplement en affectant son attribut `task.msg`, et la méthode `task.run()` retourne maintenant tous les messages que la coroutine pourrait `yield`-er.

Reste à déterminer la forme des messages grâce auxquels la coroutine peut faire appel à sa boucle d'exécution. De manière similaire aux appels système, on peut par exemple décider que ces messages seront un tuple sous la forme suivante :

```
(TYPE_APPEL, arguments)
```

Par exemple, pour demander à la boucle événementielle de lancer une ou plusieurs coroutines en parallèle, le message serait :

```
('SCHEDULE', [task1, task2, ...])
```

Modifions la méthode `_loop()` de notre boucle pour qu'elle réagisse à ces messages :

```
CALL_SCHEDULE = 'SCHEDULE'
```

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

```

def _loop(self):
    task = self._running.popleft()
    msg = task.run() # Réception du message

    if task.is_done():
        print(task)
    else:
        self.schedule(task)

    if not msg:
        return

    msg_type, args = msg
    if msg_type == CALL_SCHEDULE:
        # Si le message est un ordre d'exécuter des tâches parallèles,
        # on programme ces tâches et on les retourne à la tâche
        # appelante.
        task.msg = tuple(self.schedule(subtask) for subtask in args)
    else:
        raise RuntimeError("Message inconnu : {}".format(msg_type))

```

On peut maintenant abstraire ce message derrière une coroutine que nous appellerons `ensure_future()`, pour respecter la même nomenclature qu'`asyncio` :

```

def ensure_future(task):
    # L'appel à CALL_SCHEDULE retourne un tuple à un seul élément dans ce cas
    (task,) = yield CALL_SCHEDULE, [task]
    return task

```

Nous disposons désormais de deux façons d'exécuter une sous-tâche depuis une coroutine :

- `yield from subtask()` : lance l'exécution d'une coroutine de façon *séquentielle*, c'est-à-dire en lui laissant la main jusqu'à ce que celle-ci se soit terminée.
- `yield from ensure_future(subtask())` : lance l'exécution d'une coroutine *en parallèle*.

Ainsi, si nous modifions notre coroutine `example()` définie plus haut en conséquence, nous pouvons vérifier que nos tâches sont bien exécutées de façon concurrente :

```

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

```

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 = yield from 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...

Mais ça, je vous le garde pour le prochain exemple. ;)

Exemple n°3 : Attendre de façon asynchrone

À la fin du précédent exemple, nous avons implémenté un petit mécanisme *d'annulation* de tâches en cours d'exécution, qui permet d'éviter qu'une tâche *fille* survive à sa tâche *mère* en s'exécutant plus longtemps qu'elle. Cependant, il n'est peut-être pas toujours adéquat d'annuler brutalement une tâche. Par exemple, on peut imaginer que si celle-ci a des ressources à libérer, l'annuler risque de créer une fuite de mémoire...

Pour cette raison, on veut se donner un moyen *d'attendre* (de façon asynchrone) qu'une tâche ait fini de s'exécuter. En soi, cela n'est pas bien difficile, puisqu'il suffit de *yield-er tant que* l'événement que nous attendons ne s'est pas encore produit :

```

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

    print("En attente de la fin de 'subtask'")
    while not sub.is_done() and not sub.is_cancelled():
        yield
    print("Ça y est, je peux quitter")

```

```
def subtask():
    print("Tâche 'subtask'")
    for _ in range(5):
        print("(subtask)")
        yield
```

Essayons :

```
>>> event_loop = Loop()
>>> event_loop.run_until_complete(example())
Tâche 'example'
Lancement de la tâche 'subtask'
Retour dans 'example'
(example)
Tâche 'subtask'
(subtask)
(example)
(subtask)
(example)
(subtask)
En attente de la fin de 'subtask'
(subtask)
(subtask)
<Task 'subtask' [FINISHED] (None)>
Ça y est, je peux quitter
<Task 'example' [FINISHED] (None)>
```

C'est plutôt enfantin, en fait.

Souvenez-vous maintenant de la coroutine `wait()` d'`asyncio` que nous avons aperçue dans l'exemple n°1 :

```
>>> import asyncio
>>> loop = asyncio.get_event_loop()
>>> loop.run_until_complete(asyncio.wait([tic_tac(), spam()]))
Tic
spam
Tac
eggs
bacon
({Task(<tic_tac>)<result='Boum!>', Task(<spam>)<result='spam'>}, set())
```

Celle-ci permet d'attendre qu'une ou plusieurs tâches (lancées en parallèle) soient terminées avant de rendre la main. Sa valeur de retour se compose de deux ensembles (`set()`) :

- Le premier contient les tâches qui se sont terminées normalement ;
- Le second contient les tâches qui ont été annulées ou ont quitté sur une erreur.

C'est cette fonction que nous allons implémenter maintenant, parce qu'il serait vraiment dommage de se passer de sa souplesse d'utilisation !

En fait, le plus difficile dans cette fonction est surtout sa partie cosmétique. Pour avoir un comportement souple, il faut gérer le cas où les tâches passées à cette fonction sont des coroutines qui n'ont pas encore été lancées, ou bien des instances de la classe `Task` que la boucle événementielle aurait préalablement créées pour programmer leur exécution.

Voilà ce que cela peut donner :

```
def wait(tasks):
    # On commence par séparer les tâches en cours d'exécution des autres
    running, to_launch, finished, error = set(), set(), set(), set()
    for task in tasks:
        if isinstance(task, Task):
            if task.status == STATUS_FINISHED:
                finished.add(task)
            elif task.status in {STATUS_ERROR, STATUS_CANCELLED}:
                error.add(task)
            else:
                running.add(task)
        else:
            to_launch.add(task)

    # On lance les tâches qui ont besoin d'être lancées
    if to_launch:
        launched = yield CALL_SCHEDULE, to_launch
        running.update(launched)

    # On attend que tout le monde ait fini de s'exécuter
    while running:
        # On itère sur une copie immuable de l'ensemble 'running'
        # de façon à pouvoir modifier celui-ci sans risque dans la boucle
        for task in tuple(running):
            if not (task.is_done() or task.is_cancelled()):
                continue
            running.remove(task)
            if task.status == STATUS_FINISHED:
                finished.add(task)
            else:
                error.add(task)

        # S'il y a encore des tâches inachevées, on suspend la coroutine
        if running:
            yield

    return finished, error
```

Vérifions que nous avons bien un comportement similaire à `asyncio` :

```
>>> event_loop = Loop()
>>> event_loop.run_until_complete(wait([tic_tac(), spam()]))
Tic
```



```

Spam
Tac
Eggs
<Task 'tic_tac' [FINISHED] ('Boom!')>
Bacon
<Task 'spam' [FINISHED] ('SPAM!')>
<Task 'wait' [FINISHED] (({<Task 'tic_tac' [FINISHED] ('Boom!')>, <Task 'spam' [FINISHED] ('SP

```

Pas mal! Et pour attendre une tâche préalablement démarrée?

```

>>> def example():
...     print("Tâche 'example'")
...     print("Lancement de la tâche 'subtask'")
...     sub = yield from ensure_future(subtask())
...     print("Retour dans 'example'")
...     for _ in range(3):
...         print("(example)")
...         yield
...
...     print("En attente de la fin de 'subtask'")
...     yield from wait([sub])
...     print("Ça y est, je peux quitter")
...
>>> event_loop.run_until_complete(example())
Tâche 'example'
Lancement de la tâche 'subtask'
Retour dans 'example'
(example)
Tâche 'subtask'
(subtask)
(example)
(subtask)
(example)
(subtask)
En attente de la fin de 'subtask'
(subtask)
(subtask)
<Task 'subtask' [FINISHED] (None)>
Ça y est, je peux quitter
<Task 'example' [FINISHED] (None)>

```

Parfait. Nous venons d'implémenter notre première coroutine *d'attente asynchrone*. Celle-ci permet d'endormir une tâche pour ne la réveiller que lorsqu'une ou plusieurs autres tâches auront fini de s'exécuter. S'il y avait une seule chose à retenir de la programmation asynchrone, c'est bien ce mécanisme. C'est dans le fait de pouvoir dire à une boucle événementielle : « je fais une pause, réveille-moi quand il se sera passé quelque chose d'intéressant », que réside tout l'intérêt de ce modèle d'exécution. Et ce genre d'attente est vraiment très fréquent en informatique!

Exemple n°4 : Modélisons le serveur du *fast food* avec *asyncio*

Maintenant que nous avons compris comment fonctionne sa boucle événementielle, il est temps de nous mettre en jambes avec *asyncio* en l'utilisant pour modéliser l'exemple du début de cet article : l'employé de *fast food*.

Voici d'abord à quoi ressemble le programme dans sa version synchrone.

```
from time import sleep
from datetime import datetime

def get_burger(client):
    print("> Commande du burger pour '{}' en cuisine".format(client))
    sleep(4)
    print("< Le burger de '{}' est prêt".format(client))

def get_fries(client):
    print("> Mettre des frites à cuire pour {}".format(client))
    sleep(8)
    print("< Les frites de '{}' sont prêtes".format(client))

def get_soda(client):
    print("> Remplissage du gobelet de soda pour {}".format(client))
    sleep(2)
    print("< Le soda de '{}' est prêt".format(client))

def serve(client):
    print("Préparation de la commande de '{}'".format(client))
    start = datetime.now()
    get_burger(client)
    get_fries(client)
    get_soda(client)
    duration = datetime.now() - start
    print("Commande de '{}' prête en {}".format(client, duration))
```

Résultat : la commande est prête en 14 secondes.

```
>>> serve('A')
Préparation de la commande de 'A'
> Commande du burger pour 'A' en cuisine
< Le burger de 'A' est prêt
> Mettre des frites à cuire pour A
< Les frites de 'A' sont prêtes
> Remplissage du gobelet de soda pour A
< Le soda de 'A' est prêt
Commande de 'A' prête en 0:00:14.015165
```

Il ne nous reste plus qu'à implémenter cet exemple avec *asyncio*. Le premier jet n'est pas très difficile : il faut juste penser à utiliser la coroutine *asyncio.sleep()* plutôt que *time.sleep()* pour que l'endormissement de la tâche ne soit pas bloquant.

```

import asyncio
from datetime import datetime

@asyncio.coroutine
def get_burger(client):
    print("> Commande du burger pour {} en cuisine".format(client))
    yield from asyncio.sleep(4)
    print("< Le burger de {} est prêt".format(client))

@asyncio.coroutine
def get_fries(client):
    print("> Mettre des frites à cuire pour {}".format(client))
    yield from asyncio.sleep(8)
    print("< Les frites de {} sont prêtes".format(client))

@asyncio.coroutine
def get_soda(client):
    print("> Remplissage du gobelet de soda pour {}".format(client))
    yield from asyncio.sleep(2)
    print("< Le soda de {} est prêt".format(client))

@asyncio.coroutine
def serve(client):
    print("Préparation de la commande de {}".format(client))
    start = datetime.now()
    yield from asyncio.wait([
        get_burger(client),
        get_fries(client),
        get_soda(client),
    ])
    duration = datetime.now() - start
    print("Commande de {} prête en {}".format(client, duration))

```

Remarquez que nous avons décoré toutes nos coroutines avec le décorateur `@asyncio.coroutine` : il s'agit d'une convention pour distinguer les fonctions synchrones des coroutines asynchrones dans du code utilisant `asyncio`. En dehors de l'indice visuel, ce décorateur permet de s'assurer qu'une fonction donnée sera considérée par `asyncio` comme une coroutine, même si celle-ci ne `yield` jamais.

Exécutons-le maintenant :

```

>>> import asyncio
>>> loop = asyncio.get_event_loop()
>>> loop.run_until_complete(serve('A'))
Préparation de la commande de A
> Commande du burger pour A en cuisine
> Mettre des frites à cuire pour A
> Remplissage du gobelet de soda pour A
< Le soda de A est prêt
< Le burger de A est prêt
< Les frites de A sont prêtes

```

Commande de A prête en 0:00:08.005791

Nous sommes passés de 14 secondes à 8 secondes.

Et pour servir deux clients à la fois ?

```
>>> loop.run_until_complete(asyncio.wait([serve('A'), serve('B')]))
Préparation de la commande de B
Préparation de la commande de A
> Mettre des frites à cuire pour B
> Commande du burger pour B en cuisine
> Remplissage du gobelet de soda pour B
> Remplissage du gobelet de soda pour A
> Commande du burger pour A en cuisine
> Mettre des frites à cuire pour A
< Le soda de B est prêt
< Le soda de A est prêt
< Le burger de B est prêt
< Le burger de A est prêt
< Les frites de B sont prêtes
< Les frites de A sont prêtes
Commande de B prête en 0:00:08.005967
Commande de A prête en 0:00:08.005940
```

Les deux ont mis tout autant de temps, néanmoins cet affichage ne semble pas très réaliste.

En effet :

- la machine à soda ne peut faire couler qu'un seul soda à la fois, or ici les deux sodas ont été préparés simultanément.
- la cuisine ne comporte que 3 cuisiniers, donc on ne peut avoir au maximum que 3 hamburgers en cours de préparation à un instant donné.
- le bac à frites fait cuire en général 5 portions de frites en même temps, or ici il a été utilisé deux fois en parallèle pour produire uniquement deux portions de frites.

Comment modéliser ces contraintes ?

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 :

```
SODA_LOCK = asyncio.Lock()
```

```
@asyncio.coroutine
def get_soda(client):
    # Acquisition du verrou
    with (yield from SODA_LOCK):
        # Une seule tâche à la fois peut exécuter ce bloc
        print("> Remplissage du gobelet de soda pour {}".format(client))
        yield from asyncio.sleep(2)
        print("< Le soda de {} est prêt".format(client))
```

Le `with (yield from 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 que le soda en cours de préparation soit prêt avant de se servir de la machine.

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)
```

```
@asyncio.coroutine
def get_burger(client):
    print("> Commande du burger pour {} en cuisine".format(client))
    with (yield from BURGER_SEM):
        print("* Le burger de {} est en préparation".format(client))
        yield from asyncio.sleep(4)
        print("< Le burger de {} est prêt".format(client))
```

Le `with (yield from 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_PARTS = 0
FRIES_LOCK = asyncio.Lock()

@asyncio.coroutine
def get_fries(client):
    global FRIES_PARTS
    with (yield from FRIES_LOCK):
        print(
            "> Récupération d'une portion de frites pour {}".format(client)
        )
        if FRIES_PARTS == 0:
            print("* Mettre des frites à cuire")
            yield from asyncio.sleep(8)
            FRIES_PARTS = 5
            print("* Les frites sont prêtes")
```

```
FRIES_PARTS -= 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.

Voyons voir ce que cela donne à l'exécution :

```
>>> loop.run_until_complete(asyncio.wait([serve('A'), serve('B')]))
Préparation de la commande de B
Préparation de la commande de A
> Remplissage du gobelet de soda pour B
> Commande du burger pour B en cuisine
* Le burger de B est en préparation
> Récupération d'une portion de frites pour B
* Mettre des frites à cuire
> Commande du burger pour A en cuisine
* Le burger de A est en préparation
< Le soda de B est prêt
> Remplissage du gobelet de soda pour A
< Le burger de B est prêt
< Le burger de A est prêt
< Le soda de A est prêt
* Les frites sont prêtes
< Les frites de B sont prêtes
> Récupération d'une portion de frites pour A
< Les frites de A sont prêtes
Commande de B prête en 0:00:08.006742
Commande de A prête en 0:00:08.006859
({Task(<serve><result=None>, Task(<serve><result=None>}, set())
```

Et voilà. Nos deux tâches prennent le même temps, 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'])
... )
...
# ...
Commande de C prête en 0:00:08.009554
Commande de G prête en 0:00:08.009934
Commande de B prête en 0:00:08.010281
Commande de D prête en 0:00:08.014250
Commande de H prête en 0:00:10.017237
Commande de I prête en 0:00:16.014170
Commande de E prête en 0:00:16.014511
Commande de A prête en 0:00:16.022411
```

```
Commande de J prête en 0:00:18.023141
Commande de F prête en 0:00:20.026096
```

On se rend compte que les performances de notre serveur de fast-food se dégradent plus ou moins, même si on est toujours loin des 140 secondes que le serveur aurait pris s'il avait traité toutes ces commandes de façon synchrone.

Cela dit, il est plutôt rare que les clients passent leurs commandes tous en même temps. Une modélisation plus proche de la réalité serait que ces dix commandes arrivent à deux secondes d'intervalle :

```
@asyncio.coroutine
def test():
    tasks = []
    for client in 'ABCDEFGHIJ':
        # appel équivalent à notre fonction `launch()`
        task = asyncio.async(serve(client))
        tasks.append(task)
        yield from asyncio.sleep(2)
    yield from asyncio.wait(tasks)
```

Dans ces conditions, les temps d'attente individuels de chaque client sont assez largement réduits :

```
>>> loop = asyncio.get_event_loop()
>>> loop = asyncio.run_until_complete(test())
Préparation de la commande de A
Préparation de la commande de B
Préparation de la commande de C
Préparation de la commande de D
Commande de A prête en 0:00:08.004599
Commande de B prête en 0:00:06.002353
Préparation de la commande de E
Commande de C prête en 0:00:04.002907
Préparation de la commande de F
Commande de D prête en 0:00:04.004343
Préparation de la commande de G
Commande de E prête en 0:00:04.004445
Préparation de la commande de H
Préparation de la commande de I
Commande de F prête en 0:00:08.005227
Commande de G prête en 0:00:06.005087
Préparation de la commande de J
Commande de H prête en 0:00:04.003595
Commande de I prête en 0:00:04.003620
Commande de J prête en 0:00:04.004184
```

À raison d'un client toutes les deux secondes, notre serveur est capable de traiter des commandes avec un temps moyen de 5.2 secondes, le maximum étant 8 secondes et le minimum à 4 secondes.

Et si nous *stressions* un peu notre serveur et qu'on lui passait une commande par seconde ?

```
@asyncio.coroutine
def test():
```

```

tasks = []
for client in 'ABCDEFGHJIJ':
    task = asyncio.async(serve(client))
    tasks.append(task)
    yield from asyncio.sleep(1)
yield from asyncio.wait(tasks)

```

Comme prévu, celui-ci, sous la pression, est un peu moins efficace à cause des contraintes des différentes machines qu'il utilise, même si l'on reste tout à fait loin de l'inefficacité pathologique de la version synchrone :

```

>>> loop = asyncio.get_event_loop()
>>> loop.run_until_complete(test())
Préparation de la commande de A
Préparation de la commande de B
Préparation de la commande de C
Préparation de la commande de D
Préparation de la commande de E
Préparation de la commande de F
Préparation de la commande de G
Préparation de la commande de H
Commande de A prête en 0:00:08.003498
Commande de B prête en 0:00:07.001989
Commande de C prête en 0:00:06.000267
Commande de D prête en 0:00:05.004673
Préparation de la commande de I
Préparation de la commande de J
Commande de E prête en 0:00:06.005059
Commande de F prête en 0:00:11.002432
Commande de G prête en 0:00:10.002475
Commande de H prête en 0:00:09.000938
Commande de I prête en 0:00:10.002911
Commande de J prête en 0:00:11.004703

```

On peut se demander à quel endroit le service est ralenti quand on sert 1 client par seconde :

- S'agit-il de la machine à sodas, qui produit un soda toutes les 2 secondes ?
- S'agit-il du bac à frites, qui peut produire 5 portions en 8 secondes ?
- Ou bien s'agit-il de la cuisine, qui produit un hamburger en 4 secondes, mais reste limitée à trois cuisiniers ?

Les réponses à ces questions vous sont laissées en guise d'exercice. Vous pouvez essayer d'apporter les modifications suivantes au modèle, pour aider le gérant du restaurant à optimiser son service :

- Mettre en place une seconde machine à sodas aussi rapide que la première.
- Acheter un nouveau bac à frites pouvant produire 6 portions en 7 secondes.
- Embaucher un quatrième cuisinier.

Bon courage !

Exemple n°5 : Les entrées/sorties, le nerf de la guerre

Vous vous demandez peut-être pourquoi on parle tout le temps d'*IO* en programmation asynchrone. En effet, les entrées/sorties des programmes semblent indissociables du concept d'asynchrone, à tel point que cela se traduit jusque dans le nom de la bibliothèque standard `asyncio` de Python. Mais *pourquoi* ?

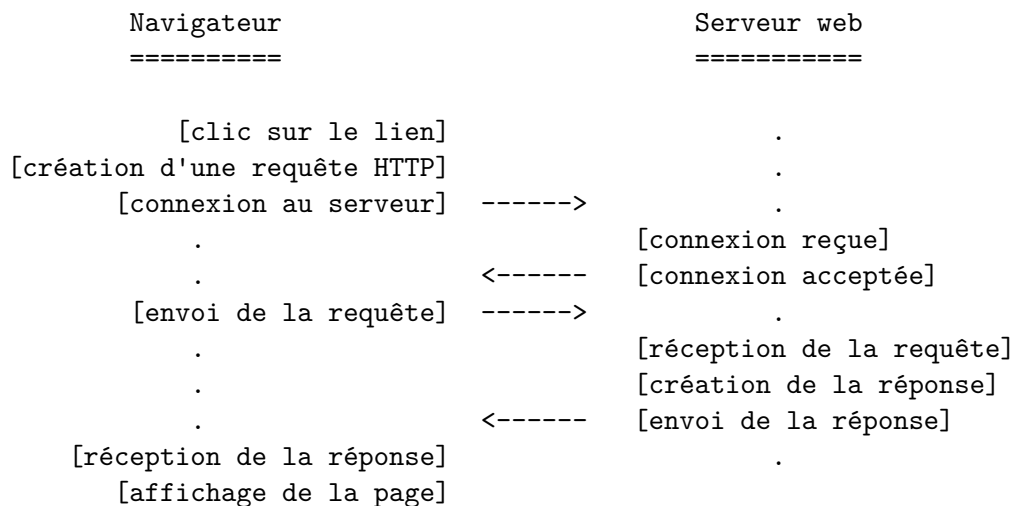
Commençons par une définition : une *IO*, c'est une opération pendant laquelle un programme *interagit avec un flux de données*. Ce **flux de données** peut être plein de choses : une connexion réseau, les flux standard `STDIN`, `STDOUT` ou `STDERR` du processus en cours d'exécution, un fichier, ou même une abstraction matérielle⁵. « Interagir avec un flux de données », ça veut dire l'**ouvrir**, **lire** ou **écrire** dedans ou le **fermer**.

Jusqu'ici, nous avons travaillé sur des exemples très simples qui se contentaient d'afficher des choses à l'écran pour bien comprendre l'ordre dans lequel les instructions étaient exécutées. Nos tâches ne réalisaient du coup que des entrées/sorties, certes, mais celles-ci étaient *synchrones* : on a considéré jusqu'à maintenant qu'un `print()` dans la console s'exécute immédiatement et sans délai lors de son appel, ce qui est parfaitement intuitif...

... mais pas toujours le reflet de la réalité.

Prenons par exemple une *IO* très simple que vous réalisez en permanence sur votre ordinateur ou smartphone sans même vous en rendre compte : **que se passe-t-il entre le moment où vous avez cliqué sur un lien dans une page web, et celui où le résultat commence à s'afficher sur votre écran** ?

Eh bien vous **attendez**. Tout simplement. Et votre navigateur aussi. Sans rentrer dans le détail du protocole HTTP, on peut schématiser grossièrement ce qui se passe comme ceci :



Dans ce schéma, tous les points (.) symbolisent une attente. Un échange HTTP (et plus généralement une *IO*), c'est une opération pendant laquelle les programmes, passent le plus clair de leur temps à **ne rien faire**. Et votre navigateur lui-même vous le dit (généralement dans un petit cadre en bas à gauche de l'écran) :

5. On peut par exemple lire un son sous Linux en *écrivant* des données dans un fichier spécial qui représente la carte son !

Dans ces conditions, l'idée de base de la programmation asynchrone est de *mettre à profit* tout ce temps que l'on passe à attendre pendant la réalisation d'une IO pour **s'occuper en faisant autre chose**.

L'exemple du serveur de *fast food* que nous avons modélisé plus tôt n'est pas anodin ; qu'il s'agisse du serveur *bien réel* d'un restaurant ou celui d'une application réseau, les deux réalisent en général des opérations comparables. En effet, de très nombreux serveurs (par exemple d'applications Web) fonctionnent plus ou moins suivant ce schéma :

1. Recevoir une requête, une commande ou un message,
2. Aller récupérer des ressources à différents endroits,
3. Combiner les ressources entre elles,
4. Répondre au client.

Dans ce schéma, les points 1, 2 et 4 peuvent être des *IO* :

1. Réception :
 - **Attente** d'une connexion,
 - Acceptation de la connexion,
 - **Attente** du message,
 - Réception du message
2. Récupérer des ressources :
 - S'il s'agit de ressources distantes :
 - Connexion à un service,
 - **Attente** de l'acceptation de la connexion,
 - Envoi d'une requête,
 - **Attente** que la réponse arrive,
 - Réception de la réponse,
 - Si la ressource est protégée par un verrou ou un sémaphore :
 - **Attente** de l'acquisition du verrou/sémaphore,
 - Récupération de la ressource,
 - Relâchement du verrou/sémaphore,
3. Combiner les ressources entre elles,
4. Répondre au client :
 - **Attente** que le *medium* soit disponible en écriture,
 - Envoi de la réponse.

Comme vous le voyez, il est vraiment *très* courant d'attendre pour un serveur. Et il ne s'agit là que d'un schéma particulier dans une infinité d'applications possibles.

Ainsi, il serait possible d'optimiser de nombreux programmes en les rendant asynchrones, pour peu que l'on soit capable de rendre leurs *IO non bloquantes*, c'est-à-dire que l'on puisse laisser la main à d'autres tâches au lieu d'attendre, et reprendre celle-ci avec l'assurance que l'on pourra réaliser une *IO* immédiatement.

Il existe sous la plupart des systèmes d'exploitation une fonctionnalité qui permet de déterminer à un instant donné si un ou plusieurs flux de données sont accessibles en lecture ou en écriture. En fait, il en

existe plein, mais nous allons nous concentrer sur celle qui sera disponible sur la plupart des systèmes d'exploitation : il s'agit de l'appel-système `select()`.

Celui-ci, en Python, se présente sous la forme suivante :

```
select.select(rlist, wlist, xlist[, timeout])
```

Où :

- `rlist` est une liste de flux sur lesquels nous voulons lire des données,
- `wlist` est une liste de flux dans lesquels nous voulons écrire des données,
- `xlist` est une liste de flux que l'on surveille jusqu'à ce qu'il se produise des *conditions exceptionnelles* (ne nous attardons pas là-dessus),
- `timeout` est une durée (optionnelle) pendant laquelle on attend que des flux soient disponibles. Par défaut, on attend indéfiniment. Si on lui passe la valeur 0, l'appel à `select()` retourne immédiatement.

Cette fonction retourne trois listes :

- une contenant les flux disponibles en lecture à l'instant T,
- une contenant les flux disponibles en écriture à l'instant T,
- une autre contenant les flux victimes d'une exception.

Que demander de plus ? Nous avons à notre disposition une fonction, dont l'exécution est instantanée, qui pourra nous permettre de réveiller les tâches en attente de lecture ou d'écriture.

On peut donc implémenter les coroutines d'attente asynchrones suivantes :

```
from select import select
```

```
# Attendre de façon asynchrone qu'un flux soit disponible en lecture
```

```
def wait_readable(stream):  
    while True:  
        rlist, _, _ = select([stream], [], [], 0)  
        if rlist:  
            return stream  
        yield
```

```
# Attendre de façon asynchrone qu'un flux soit disponible en écriture
```

```
def wait_writable(stream):  
    while True:  
        _, wlist, _ = select([], [stream], [], 0)  
        if wlist:  
            return stream  
        yield
```

Note : Sous la plupart des systèmes d'exploitation Unix, vous pouvez utiliser `select()` pour attendre après n'importe quel flux de données (flux standard, fichiers, sockets), mais sous Windows, *seules* les sockets sont supportées.

Pour bien comprendre l'apport de ces deux fonctions, commençons par écrire un petit serveur qui se contente d'attendre une seconde avant de renvoyer les messages des clients :

```

from socket import socket
from time import sleep

def echo_server():
    # On crée une socket (par défaut : TCP/IP)
    # qui écoutera sur le port 1234
    sock = socket()
    sock.bind(('localhost', 1234))

    # On garde un maximum de 5 demandes de connexion en attente
    sock.listen(5)
    try:
        while True:
            # Acceptation de la connexion
            conn, host = sock.accept()

            # Réception d'un message (4 Mio max.)
            msg = conn.recv(4096)
            print("message reçu de {!r}: {}".format(host, msg.decode()))
            sleep(1)

            # Renvoi du message
            conn.send(msg)
            conn.close()
    finally:
        sock.close()

```

Lançons ce serveur dans une console.

```
>>> echo_server()
```

Dans **une autre console**, nous pouvons vérifier que celui-ci fonctionne en lui envoyant un message.

```

>>> sock = socket.socket()
>>> sock.connect(('localhost', 1234))
>>> sock.send("Ohé !".encode())
6

```

Le serveur affiche alors :

```
message reçu de ('127.0.0.1', 40568): Ohé !
```

Nous n'avons plus qu'à récupérer sa réponse dans la fenêtre du client :

```

>>> data = sock.recv(1024)
>>> data.decode()
'Ohé !'

```

Parfait.

Pour corser les choses, essayons maintenant de lui envoyer 5 messages à la fois. Utilisons pour cela `asyncio`, ainsi que les deux coroutines d'attente que nous venons d'écrire :

```

from socket import socket
from datetime import datetime

@asyncio.coroutine
def echo(msg):
    # Connection TCP au port 1234
    sock = socket()
    sock.connect(('localhost', 1234))

    # On attend de pouvoir envoyer le message
    yield from wait_writable(sock)
    print("Envoi du message {!r}".format(msg))
    sock.send(msg.encode())

    # On attend la réponse
    yield from wait_readable(sock)
    data = sock.recv(4096)
    print("Message reçu:", data.decode())
    sock.close()

@asyncio.coroutine
def echo_client():
    start = datetime.now()
    tasks = [echo("Hello {}".format(num)) for num in range(5)]
    yield from asyncio.wait(tasks)
    print("temps total:", datetime.now() - start)

```

Rien de fondamentalement compliqué : la coroutine `echo()` se contente de faire ce que nous avons réalisé juste avant dans la console, mais en attendant de façon asynchrone que la socket soit disponible en écriture, puis en lecture.

Voici le résultat :

```

>>> loop = asyncio.get_event_loop()
>>> loop.run_until_complete(echo_client())
Envoi du message 'Hello 2'
Envoi du message 'Hello 1'
Envoi du message 'Hello 0'
Envoi du message 'Hello 3'
Envoi du message 'Hello 4'
Message reçu: Hello 2
Message reçu: Hello 1
Message reçu: Hello 0
Message reçu: Hello 3
Message reçu: Hello 4
temps total: 0:00:05.013461

```

À l'exécution, nous nous rendons compte que le serveur, qui est *synchrone*, traite les messages les uns à la suite des autres. Réimplémentons-le avec `asyncio` :

```

from socket import socket

```

```

import asyncio

@asyncio.coroutine
def serve_echo(conn, host):
    # On suspend l'exécution jusqu'à recevoir un message
    yield from wait_readable(conn)
    msg = conn.recv(4096)
    print("message reçu de {!r}: {}".format(host, msg.decode()))
    yield from asyncio.sleep(1)

    # On suspend l'exécution jusqu'à pouvoir répondre au client
    yield from wait_writable(conn)
    conn.send(msg)
    conn.close()

@asyncio.coroutine
def echo_server():
    # On crée une socket (par défaut : TCP/IP)
    # qui écoutera sur le port 1234
    sock = socket()
    sock.bind(('localhost', 1234))

    # On garde un maximum de 5 demandes de connexion en attente
    sock.listen(5)
    try:
        while True:
            # On attend qu'une demande de connexion arrive
            yield from wait_readable(sock)

            # Acceptation de la connexion
            conn, host = sock.accept()

            # On programme le traitement de la requête dans une tâche séparée.
            asyncio.async(serve_echo(conn, host))
    finally:
        sock.close()

```

Lançons le serveur :

```

>>> loop = asyncio.get_event_loop()
>>> loop.run_until_complete(echo_server())

```

Puis le client :

```

>>> loop.run_until_complete(echo_client())
Envoi du message 'Hello 3'
Envoi du message 'Hello 0'
Envoi du message 'Hello 2'
Envoi du message 'Hello 1'
Envoi du message 'Hello 4'

```

```
Message reçu: Hello 3
Message reçu: Hello 0
Message reçu: Hello 2
Message reçu: Hello 1
Message reçu: Hello 4
temps total: 0:00:01.001991
```

Cette fois-ci, le serveur a traité toutes les requêtes en même temps, pour un temps d'exécution total de 1s au lieu de 5s.

Nous voici maintenant capables de réaliser des *IO* asynchrones! Néanmoins, bien que nos coroutines `wait_readable()` et `wait_writable()` soient tout à fait fonctionnelles dans cet exemple, je vous **déconseille très vivement** de les utiliser. En effet, celles-ci bouclent à l'infini jusqu'à ce que le flux soit disponible, au lieu d'attendre passivement sans consommer de temps sur le processeur.

Rassurez-vous, `asyncio` propose de nombreuses façons de réaliser la même chose, et ce sans surcharger le processeur. Nous les examinerons dans les exemples suivants.

Exemple n°6 : Des applications réseau avec `asyncio`

Le précédent exemple nous a fait toucher du doigt le concept d'*IO asynchrones*. Il est temps pour nous de nous familiariser avec les objets que nous propose `asyncio` pour réaliser de telles opérations sur un flux réseau.

Commençons par examiner un exemple que nous détaillerons ensuite. Voici le client du précédent exemple, réimplémenté avec les outils d'`asyncio` :

```
@asyncio.coroutine
def echo_client(message):
    reader, writer = yield from asyncio.open_connection('localhost', 1234)

    print("Envoi du message :", message)
    writer.write(message.encode())

    data = yield from reader.read(1024)
    print("Message reçu :", data.decode())

    writer.close()
```

Même sans savoir ce que sont ces objets `reader` et `writer`, la lecture de ce code est plutôt intuitive :

- La coroutine `asyncio.open_connection()` crée une connexion et retourne deux objets :
 - Un “*reader*” que l’on peut utiliser pour recevoir des données,
 - Un “*writer*” dont on se sert pour en envoyer,
- On se sert du `writer` pour envoyer un message au serveur (attention : la méthode `write()` est une fonction et non une coroutine ; pas de `yield from`),
- On appelle ensuite, cette fois de façon asynchrone, la méthode `reader.read()` pour récupérer des données (on s’attend à moins de 1024 octets),
- On ferme le `writer` pour clore la connexion.

Avant d’aller plus loin, lançons le serveur de l’exemple précédent et vérifions que ce code fonctionne dans la console :

```
>>> import asyncio
>>> loop = asyncio.get_event_loop()
>>> loop.run_until_complete(echo_client("Coucou!"))
Envoi du message : Coucou!
Message reçu : Coucou!
```

Par de surprise, le code se comporte comme prévu. Attardons-nous un peu sur ces fameux objets qu’`asyncio` a créés pour nous. Il s’agit en fait d’instances de `asyncio.StreamReader` et `asyncio.StreamWriter`.

Il s’agit de l’interface haut niveau d’`asyncio` pour manipuler une connexion de type “*Stream*” (typiquement, TCP). La même interface existe également pour les sockets Unix. Ces objets représentent un les deux sens d’un même flux réseau. Ainsi, vous avez grosso-modo 4 méthodes à retenir :

- La *coroutine* `StreamReader.read(size)` sert à recevoir un bloc de `size` octets de données maximum, de façon asynchrone;
- La *coroutine* `StreamReader.readline()` sert à recevoir une “ligne” de données, terminée par le caractère ASCII spécial `LINE_FEED` (`'\n'`);
- La *fonction* `StreamWriter.write(data)` sert à envoyer des données (sous la forme d’un objet `bytes`) sur le flux. En fait, les données seront placées en attente dans un tampon (*buffer*) qui ne sera vidé qu’au prochain `yield...`
- Ce qui nous amène à la *coroutine* `StreamWriter.drain()`, qui sert à vider explicitement le tampon du *writer* pour envoyer les données sur le flux.

C’est plutôt simple, non ? Ces objets reposent en fait sur une autre API, un petit peu plus bas niveau, d’`asyncio`, qui modélise des *protocoles* que l’on utilise par-dessus un *transport* (TCP, UDP...). Nous ne parlerons pas de cette API dans cet exemple, mais si vous êtes curieux, [sa documentation](#) est plutôt claire et détaillée.

Cette même interface peut d’ailleurs nous permettre de créer facilement un serveur, au moyen de la *coroutine* `asyncio.start_server()` :

```
#!/usr/bin/env python3
import asyncio

@asyncio.coroutine
def handle_echo(reader, writer):
    data = yield from reader.read(1024)
    print("Message reçu :", data.decode())
    yield from asyncio.sleep(1)
    writer.write(data)
    yield from writer.drain()
    writer.close()

def echo_server():
    loop = asyncio.get_event_loop()
    server = loop.run_until_complete(
        asyncio.start_server(handle_echo, 'localhost', 1234)
```



```

)

try:
    loop.run_forever()
except KeyboardInterrupt:
    pass

server.close()
loop.run_until_complete(server.wait_closed())
loop.close()

if __name__ == '__main__':
    echo_server()

```

Ici, tout le code de notre serveur se trouve dans la coroutine `handle_echo`. Cette coroutine accepte un `StreamReader` et un `StreamWriter`, et ne retourne que lorsque l'échange avec un client est terminé.

Dans la fonction `echo_server()` on commence par créer un serveur en passant cette coroutine à `asyncio.start_server()` dans la boucle événementielle, puis on demande à la boucle de tourner “à l'infini” (jusqu'à ce que le processus soit interrompu).

Notez que nous aurions pu instancier le même service, mais sur des sockets Unix au lieu d'un flux TCP, simplement en utilisant la fonction `asyncio.start_unix_server()` à la place de `asyncio.start_server()` : la coroutine `handle_echo`, restera la même, et sera lancée dans une nouvelle tâche chaque fois qu'un client se connectera.