‘N’ exemples pour découvrir la programmation asynchrone

Arnaud Calmettes (nohar)

14 septembre 2015

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.

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. Pour faire avancer la tâche jusqu'au prochain `yield`, il suffit d'itérer dessus au moyen de la fonction standard `next()` :

```
>>> task = tic_tac()
>>> next(task)
```

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.

```
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 événementielle pour exécuter cette coroutine. Il suffit en fait de la considérer comme une file de tâches à exécuter jusqu'à épuisement.

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, puisqu'il s'agit d'une file d'attente, il faut 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()`.

```
>>> from collections import deque
>>> events_loop = deque()
```

Dotons-nous de fonctions utilitaires pour ajouter des tâches à la boucle événementielle et pour exécuter cette dernière :

— `schedule(loop, task)` : programmer l'exécution d'une tâche ;
— `run_until_empty(loop)` : exécuter la boucle événementielle jusqu'à épuisement ;
— `run_once(loop, task)` : raccourcis pour exécuter une coroutine.

```
def schedule(loop, task):
    loop.append(task)

def run_until_empty(loop):
    while loop:
        task = loop.popleft()
        try:
            # On fait avancer la coroutine jusqu'au prochain "yield"
            next(task)
            # Si celle-ci n'a pas fini son travail,
            # on la programme pour qu'elle le reprenne plus tard.
            schedule(loop, task)
        except StopIteration as res:
            # La coroutine a terminé son exécution.
            # On affiche sa valeur de retour.
            print(
```

```
                "Task {!r} returned {!r}".format(task.__name__, res.value)
            )

def run_once(loop, task):
    schedule(loop, task)
    run_until_empty(loop)
```

Nous pouvons maintenant nous servir de la boucle événementielle pour exécuter la tâche `tic_tac` :

```
>>> run_once(events_loop, tic_tac())
Tic
Tac
Task 'tic_tac' returned 'Boum!'
```

Tout fonctionne comme prévu. Et si nous donnions deux coroutines différentes à exécuter à boucle événementielle ?

```
>>> def spam():
...     print("spam")
...     yield
...     print("eggs")
...     yield
...     print("bacon")
...     yield
...     return 'spam'
...
>>> schedule(events_loop, tic_tac())
>>> schedule(events_loop, spam())
>>> run_until_empty(events_loop)
Tic
spam
Tac
eggs
Task 'tic_tac' returned 'Boum!'
bacon
Task 'spam' returned 'spam'
```

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

Ce genre de boucle événementielle existe bien évidemment dans Python, c'est même exactement **le centre nerveux** du module `asyncio`. Regardez :

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

Dans tous les exemples qui suivent, nous allons nous attarder sur les différentes fonctionnalités qui font d'`asyncio` un puissant *framework* de programmation asynchrone. Nous reprogrammerons certaines de ces fonctionnalités pour mieux les comprendre au début, puis nous glisserons progressivement vers les objets de plus haut niveau que nous propose ce *framework* pour nous concentrer sur des applications **pratiques** de la vie réelle.

Commençons par modéliser un peu mieux notre système. Jusqu'ici, il se compose uniquement de deux types d'objets : une **boucle** (`Loop`) qui exécute des **tâches** (`Task`) de façon concurrente.

Définissons notre classe `Task` en premier :

```
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.msg = None  # Message à envoyer à 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:
            self.status = STATUS_RUNNING
            # Cette ligne revient à faire la même chose que next(self.coro).
            # Nous y reviendrons dans le prochain exemple.
            return self.coro.send(self.msg)
        except StopIteration as err:
            self.status = STATUS_FINISHED
            self.return_value = err.value
        except Exception as err:
            self.status = STATUS_ERROR
            self.error_value = err

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

    def __repr__(self):
        return "<Task '{name}' [{status}] ({res!r})>".format(
            name=self.name,
            status=self.status,
```

```
            res=(self.return_value or self.error_value)
        )
```

Cette classe a une utilisation très simple. On lui passe une coroutine, et on se contente de l'appeler via sa méthode `run()` :

```
>>> task = Task(tic_tac())
>>> task
<Task 'tic_tac' [NEW] (None)>
>>> while not task.is_done():
...     task.run()
...     print(task)
...
Tic
<Task 'tic_tac' [RUNNING] (None)>
Tac
<Task 'tic_tac' [RUNNING] (None)>
<Task 'tic_tac' [FINISHED] ('Boum!')>
```

Reste la boucle événementielle `Loop` :

```
from collections import deque

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

    def run_once(self):
        while self._running:
            task = self._running.popleft()
            task.run()

            if task.is_done():
                print(task)
                continue

            self.schedule(task)

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

Rien que nous n'ayons pas déjà vu :

```
>>> loop = Loop()
>>> loop.schedule(tic_tac())
>>> loop.schedule(spam())
>>> loop.run_once()
Tic
spam
```

```
Tac
eggs
<Task 'tic_tac' [FINISHED] ('Boum!')>
bacon
<Task 'spam' [FINISHED] ('spam')>
```

Voilà, nous pouvons commencer à nous amuser avec nos coroutines. :)

Exemple n°2 : Communiquer avec la boucle événementielle

Jusqu'ici, vous vous êtes probablement aperçu que nos coroutines ne `yield`-aient aucune valeur. En effet, le simple fait qu'elles se suspendent suffit à les faire exécuter en concurrence. Néanmoins, nous avons vu dans un précédent article que ce mot-clé `yield` était en Python un véritable canal de communication entre la coroutine et le code qui l'exécute.

Ainsi, nous savons que nous pouvons faire communiquer la boucle avec les tâches qu'elle exécute. C'est d'ailleurs tout l'intérêt de la ligne suivante dans la classe `Task` :

```
class Task:
    def run(self):
        # ...
        return self.coro.send(self.msg)
```

Mais à quoi cela pourrait-il bien servir ? Faisons, si vous le voulez bien un petit parallèle entre notre boucle événementielle et un système d'exploitation (OS).

Vous avez peut-être déjà lu ou entendu que l'OS est chargé d'exécuter des programmes sur l'ordinateur : à chaque instant il sait quel programme est en train de s'exécuter, il a d'ailleurs le pouvoir de les démarrer et de les interrompre, exactement comme notre boucle événementielle avec ses tâches.

Parfois (enfin, plutôt souvent, même) un programme `A` a besoin que l'OS lui rende un service, par exemple pour lancer un autre programme `B`. Dans ce cas, le programme `A` va réaliser ce que l'on appelle un *appel système*, c'est-à-dire qu'il va *envoyer un message* à l'OS pour lui demander de bien vouloir lancer le nouveau programme `B`.

Eh bien nous pourrions tout à fait transposer cette notion d'*appels système* à notre boucle événementielle. Et nous allons même nous servir de `yield` pour ce faire.

Mettons nous en situation. Nous voulons que notre coroutine `example` lance une nouvelle tâche `subtask`.

La façon la plus simple de nous y prendre serait d'appeler simplement la seconde tâche dans la première, au moyen de `yield from` :

```
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(3):
        print("(subtask)")
        yield
```

Voilà ce que cela donnerait :

```
>>> loop = Loop()
>>> loop.schedule(example())
>>> loop.run_once()
Tâche 'example'
Lancement de la tâche 'subtask'
Tâche 'subtask'
(subtask)
(subtask)
(subtask)
Retour dans 'example'
(example)
(example)
(example)
<Task 'example' [FINISHED] (None)>
```

La tâche `subtask` a été exécutée complètement avant que `example` ne reprenne la main. C'est pas mal, ça nous montre comment appeler des *sous-coroutines*, mais comment faire si on avait voulu que la coroutine `subtask` s'exécute en parallèle de `example` ?

Eh bien pour cela, il faut créer une tâche dédiée à la coroutine `subtask` dans la boucle événementielle. On peut donc imaginer envoyer un message à la boucle événementielle avec le mot-clé `yield`. Ce message serait composé de deux éléments :

— une *action* (l'ordre que l'on envoie à la boucle, ici "exécuter")
— et une *valeur* (ici, la coroutine à exécuter dans une nouvelle tâche)

Notre tâche `example` ressemblerait alors à ceci :

```
def example():
    print("Tâche 'example'")
    print("Lancement de la tâche 'subtask'")
    yield 'EXEC', subtask()  # Envoi du message
    print("Retour dans 'example'")
    for _ in range(3):
        print("(example)")
        yield
```

Ce message sera retourné à la boucle lorsquelle appellera `task.run()`. Il nous suffit de rajouter un peu de code à ce niveau pour le lire :

```
MSG_EXEC = 'EXEC'

class Loop:
    def run_once(self):
```

```
        while self._running:
            task = self._running.popleft()
            msg = task.run()

            if task.is_done():
                print(task)
                continue

            self.schedule(task)

            if msg:
                # Si la coroutine a envoyé un message à la boucle
                action, value = msg
                if action == MSG_EXEC:
                    # Exécute une coroutine dans une tâche dédiée
                    self.schedule(value)
```

Et voilà le résultat :

```
>>> loop = Loop()
>>> loop.schedule(example())
>>> loop.run_once()
Tâche 'example'
Lancement de la tâche 'subtask'
Retour dans 'example'
(example)
Tâche 'subtask'
(subtask)
(example)
(subtask)
(example)
(subtask)
<Task 'example' [FINISHED] (None)>
<Task 'subtask' [FINISHED] (None)>
```

Nous pouvons donc créer notre premier "*message système*" pour masquer les détails d'implémentation à l'utilisateur :

```
def launch(task):
    if not isinstance(task, Task):
        task = Task(task)
    yield MSG_EXEC, task
    return task

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

```
        print("(example)")
        yield
```

Nous avons maintenant deux façons d'exécuter une nouvelle coroutine :

— `yield from coroutine()` suspend l'exécution de la tâche en cours jusqu'à ce que la coroutine appelée ait terminé son exécution.
— `yield from launch(coroutine())` lance la coroutine dans une tâche séparée pour que celle-ci s'exécute *en concurrence* et reprend aussitôt l'exécution de la tâche en cours.

La fonction équivalente à `launch()` dans `asyncio` est `asyncio.async()`. Notez toutefois que cette fonction est dépréciée (à cause de son nom) dans Python 3.5, qui recommande d'utiliser maintenant `asyncio.ensure_future()`.

Ces deux fonctions retournent la tâche encapsulée dans un objet appelé "futur" (`Future`), correspondant grosso-modo à notre classe `Task`.

Exemple n°3 : Attendre la fin de l'exécution d'une ou plusieurs tâches

Dans l'exemple n°1, nous avons vu un bref appel à une fonction d'`asyncio` :

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

La fonction intéressante de cet exemple est `asyncio.wait()`. Celle-ci permet d'attendre qu'une ou plusieurs tâches (lancées en parallèle) soient terminées. 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 ne sont pas encore terminées, ont été annulées ou on quitté sur une erreur.

C'est cette fonction que nous allons implémenter maintenant.

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 et non des instances de la classe `Task`,
— N'ont pas encore été programmées pour être exécutées par la boucle,
— Sont déjà en train de s'exécuter.

Voilà ce que cela peut donner :

```
def wait(tasks):
    # On lance les tâches qui ne l'ont pas encore été
    for idx, task in enumerate(tasks):
        # Si c'est une coroutine, on crée la tâche correspondante
        if not isinstance(task, Task):
            task = tasks[idx] = Task(task)
        # Si la tâche n'a pas encore été lancée, on la lance
        if task.status == STATUS_NEW:
            yield from launch(task)

    # On attend que toutes les tâches soient terminées
    while not all(task.is_done() for task in tasks):
        yield

    # On crée les deux ensembles pour le résultat
    finished = set()
    error = set()
    for task in tasks:
        if task.status == STATUS_FINISHED:
            finished.add(task)
        elif task.status == STATUS_ERROR:
            error.add(task)

    return finished, error
```

Nous pouvons vérifier que cette nouvelle fonction marche comme prévu avec les coroutines suivantes :

```
>>> def repeat(msg, times):
...    print("lancement de la tâche", repr(msg))
...     for _ in range(times):
...         print(msg)
...         yield
...
>>> def main_task():
...     print("Lancement de 3 tâches en parallèle")
...     returned, error = yield from wait([
...         repeat("spam", 10),
...         repeat("eggs", 3),
...         repeat("bacon", 5),
...     ])
...     print("Retour à la fonction principale: {} OK, {} erreurs".format(
...         len(returned), len(error)
...     ))
...
```

Et le résultat :

```
>>> loop = Loop()
>>> loop.schedule(main_task())
>>> loop.run_once()
```

```
Lancement de 3 tâches en parallèle
lancement de la tâche 'spam'
spam
lancement de la tâche 'eggs'
eggs
spam
lancement de la tâche 'bacon'
bacon
eggs
spam
bacon
eggs
spam
bacon
<Task 'repeat' [FINISHED] (None)>
spam
bacon
spam
bacon
spam
<Task 'repeat' [FINISHED] (None)>
spam
spam
spam
<Task 'repeat' [FINISHED] (None)>
Retour à la fonction principale: 3 OK, 0 erreurs
<Task 'main_task' [FINISHED] (None)>
>>>
```

En guise d'exercice, si cela vous intéresse, vous pouvez :

- Essayer d'expliquer pourquoi les tâches ne démarrent pas toutes en même temps,
- Constater que si vous réimplémentez cet exemple avec `asyncio`, ce problème ne se produit pas,
- Modifier la fonction `wait()` et la boucle événementielle pour coller au comportement d'`asyncio`.

En ce qui nous concerne, la fonction `wait` que nous venons d'implémenter nous suffira largement pour la suite.

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 'ABCDEFGHIJ':
        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 matérielle [2]. « 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.

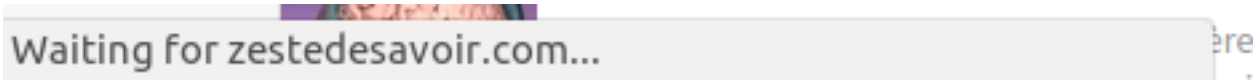
Prenons par exemple une IO très simple que vous réalisez en permanence sur votre ordinateur ou smartphone sans même vous en rendre compte : **que se passe-t-il entre le moment où vous avez cliqué sur un lien dans une page web, et celui où le résultat commence à s'afficher sur votre écran** ?

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 :

```
        Navigateur                          Serveur web
        ==========                          ===========

           [clic sur le lien]                     .
[création d'une requête HTTP]                     .
       [connexion au serveur]  ------>            .
            .                                [connexion reçue]
            .                  <------    [connexion acceptée]
        [envoi de la requête]  ------>            .
            .                                [réception de la requête]
            .                                [création de la réponse]
            .                  <------    [envoi de la réponse]
    [réception de la réponse]                     .
       [affichage de la page]
```

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) :



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 :

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

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 instantannée, 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 connection
            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 attent qu'une demande de connexion arrive
            yield from wait_readable(sock)

            # Acceptation de la connection
            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 `ayncio.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 article, mais si vous êtes curieux, sa documentation est plutôt claire et détaillée.