ENTWANNE

Zeste de savoir

Plongée au cœur de l'asynchrone en Python

26 novembre 2020

Table des matières

Le modèle asynchrone a pris beaucoup d'ampleur dans les dernières versions de Python. La bibliothèque *asyncio* a été ajoutée en Python 3.4, ont suivi les mots-clés `async` et `await`en Python 3.5, et d'autres nouveautés dans les versions suivantes.

Grâce à nohar vous savez déjà comment fonctionnent les coroutines et la programmation asynchrone en Python . Mais vous êtes-vous déjà demandé comment Python gérait cela ?

Dans ce tutoriel, j'aimerais vous faire découvrir ce qui se cache derrière les mots-clés `async` et `await`, comment ils s'interfacent avec *asyncio*. Mais aussi de quoi est faite cette bibliothèque et comment on pourrait la réécrire.

Cet article présuppose une version de Python supérieure ou égale à 3.5. Une connaissance minimale du modèle asynchrone de Python et de la bibliothèque *asyncio* sont préférables.

1. Un monde de coroutines

Depuis Python 3.5, une coroutine se définit à l'aide des mots-clés `async def` :

```
1 async def simple_print(msg):
2     print(msg)
```

Techniquement, `simple_print` n'est pas une coroutine. C'est en fait une fonction qui renvoie une nouvelle coroutine à chaque appel. Comme toute fonction, `simple_print` peut donc recevoir des arguments, qui seront utilisés par la coroutine et influeront sur son comportement.

```
>>> simple_print
<function simple_print at 0x7f0873895950>
>>> simple_print('Hello')
<coroutine object simple_print at 0x7f08738959e0>
```

Le contenu d'une coroutine ne s'exécute pas directement, il faut la lancer dans un environnement asynchrone. Par exemple avec un `await` utilisé depuis une autre coroutine.

Ici nous allons faire appel à *asyncio*, le moteur asynchrone de la bibliothèque standard. Il possède une méthode `run` permettant d'exécuter le contenu d'une coroutine.

```
>>> import asyncio
>>> asyncio.run(simple_print('Hello'))
Hello
```

Derrière cette simple ligne, *asyncio* se charge d'instancier une nouvelle boucle événementielle, de démarrer notre coroutine et d'attendre que celle-ci se termine. Si l'on omet les opérations de finalisation qu'ajoute `asyncio.run`, le code précédent est équivalent à :

```
>>> loop = asyncio.new_event_loop()
>>> asyncio.set_event_loop(loop)
>>> loop.run_until_complete(simple_print('Hello'))
Hello
```

Il s'agit donc d'une boucle événementielle, chargée d'exécuter et de cadencer les différentes tâches. La boucle est propre au moteur asynchrone utilisé, et permet une utilisation concurrente des tâches.

Mais de quoi est donc faite une coroutine ? Comment fait ce `run_until_complete` pour exécuter notre code ?

En inspectant l'objet renvoyé par `simple_print`, on remarque qu'il possède une méthode `__await__`.

```
>>> coro = simple_print('Hello')
>>> dir(coro)
['__await__', ...]
```

La coroutine serait donc un objet avec une méthode spéciale `__await__`. Nous voilà un peu plus avancés, plus qu'à en apprendre davantage sur cette méthode.

On voit qu'elle s'appelle sans arguments et qu'elle renvoie un objet de type `coroutine_wrapper`. Mais en inspectant à nouveau, on remarque que cet objet est un itérateur !

```
>>> aw = coro.__await__()
>>> aw
<coroutine_wrapper object at 0x7fcde8f30710>
>>> dir(aw)
[..., '__iter__', ..., '__next__', ..., 'send', 'throw']
```

Plus précisément, il s'agit ici d'un générateur, reconnaissable aux méthodes `send` et `throw`.

En résumé, les coroutines possèdent donc une méthode `__await__` qui renvoie un itérateur. Cela semble logique si vous vous souvenez des articles donnés en introduction, qui montrent que la coroutine est un enrobage autour d'un générateur.

Les coroutines pouvant être converties en itérateurs, on comprend maintenant comment la boucle événementielle est capable de les parcourir. Une simple boucle `for` pourrait faire l'affaire, en itérant manuellement sur l'objet renvoyé par `__await__`.

```
>>> for _ in simple_print('Hello').__await__():
...     pass
...
Hello
```

La coroutine présentée ici ne réalise aucune opération asynchrone, elle ne fait qu'afficher un message. Voici un exemple plus parlant d'une coroutine plus complexe faisant appel à d'autres tâches.

```
async def complex_work():
    await simple_print('Hello')
    await asyncio.sleep(0)
    await simple_print('World')
```

Le comportement est le même : itérer sur l'objet renvoyé par `__await__` permet d'exécuter le corps de la coroutine.

```
>>> for _ in complex_work().__await__():
...     pass
...
Hello
World
```

Mais avec cette simple boucle on ne voit pas clairement ce qui délimite les itérations. Impossible de savoir en voyant le code précédent combien la boucle a fait d'itérations (et donc à quel moment elle a repris la main).

Les itérateurs ne s'utilisent pas uniquement avec des boucles `for`, on peut aussi les parcourir pas à pas à l'aide de la fonction `next`. `next` renvoie à chaque appel l'élément suivant de l'itérateur, et lève une exception `StopIteration` en fin de parcours. C'est donc cette fonction que nous allons utiliser pour l'exécution, qui rendra visible chaque interruption de la tâche.

```
>>> it = complex_work().__await__()
>>> next(it)
Hello
>>> next(it)
World
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Cela apparaît très clairement maintenant, notre boucle réalise deux itérations. Chaque interruption permet à la boucle de reprendre la main, de gérer les événements et de cadencer les tâches (choisir de les suspendre ou de les continuer), c'est ainsi qu'elle peut en exécuter plusieurs «simultanément» (de façon concurrente).

C'est ici l'expression `await asyncio.sleep(0)` qui est responsable de l'interruption dans notre itération, elle est similaire à un `yield` pour un générateur. `await` est l'équivalent du `yield from`, il délégue l'itération à une sous-tâche. Il ne provoque pas d'interruption en lui-même, celle-ci ne survient que si elle est déclenchée par la sous-tâche (nous verrons par la suite par quel moyen).

`asyncio.sleep(0)` est un cas particulier de `sleep` qui ne fait qu'une simple interruption, sans attente. Le comportement serait différent avec une durée non nulle en paramètre.

2. Attendez-moi !

2.1. À la découverte des tâches asynchrones

Les coroutines ne sont pas les seuls objets qui peuvent s'utiliser derrière le mot-clé `await`. Plus généralement on parle de tâches asynchrones (ou *awaitables*) pour qualifier ces objets.

Ainsi, un *awaitable* est un objet caractérisé par une méthode `__await__` renvoyant un itérateur. Les coroutines sont un cas particulier de tâches asynchrones construites autour d'un générateur (avant Python 3.5, on créait d'ailleurs une coroutine à l'aide d'un décorateur—`asyncio.coroutine`—appliqué à une fonction génératrice).

Voici par exemple un équivalent à notre fonction `complex_work`. `ComplexWork` est ici une classe dont les instances sont des tâches asynchrones.

```
class ComplexWork:
    def __await__(self):
        print('Hello')
        yield
        print('World')
```

Avec le mot-clé `yield`, notre méthode `__await__` devient une fonction génératrice et renvoie donc un itérateur. On utilise `yield` sans paramètre, les valeurs renvoyées lors de l'itération ne nous intéressent pas pour l'instant, seule l'exécution importe.

Nous pouvons exécuter notre tâche asynchrone dans une boucle événementielle *asyncio* :

```
>>> loop.run_until_complete(ComplexWork())
Hello
World
```

Et notre objet respecte le protocole établi : il est possible d'itérer sur le retour d'`__await__`.

```
>>> it = ComplexWork().__await__()
>>> next(it)
Hello
>>> next(it)
World
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

2.2. Synchronisation entre tâches

En pratique, il est assez peu fréquent d'avoir besoin de définir un *awaitable* autre qu'une coroutine. C'est néanmoins utile si l'on souhaite conserver un état associé à notre tâche, pour pouvoir interagir avec elle depuis l'extérieur en altérant cet état.

Prenons par exemple la classe `Waiter` qui suit, qui permet d'attendre un résultat.

```
class Waiter:
    def __init__(self):
        self.done = False

    def __await__(self):
        while not self.done:
            yield
```

Le principe est relativement simple : l'objet est initialisé avec un état booléen `done` à `False` et son générateur (`__await__`) s'interrompt continuellement tant que l'état ne vaut pas `True`. Cela bloque la tâche asynchrone appelante puisque la boucle événementielle itérera sur un générateur infini en attendant son changement d'état.
Une fois que cet état passe à `True`, le générateur prend fin et la tâche asynchrone est donc terminée. Ça permet alors à la boucle événementielle de reprendre l'exécution à la suite de cette tâche.

On utilise `Waiter` pour synchroniser deux tâches asynchrones. En effet, avec un objet *waiter* partagé entre deux tâches, une première peut attendre sur cet objet tandis qu'une seconde exécute un calcul avant de changer l'état du *waiter* (signalant que le calcul est terminé et permettant à la première tâche de continuer).

```
async def wait_job(waiter):
    print('start')
    await waiter # wait for count_up_to to be finished
    print('finished')

async def count_up_to(waiter, n):
    for i in range(n):
        print(i)
        await asyncio.sleep(0)
    waiter.done = True
```

```
>>> waiter = Waiter()
>>> loop.run_until_complete(asyncio.gather(
...     wait_job(waiter),
...     count_up_to(waiter, 10),
... ))
start
0
1
2
3
4
5
6
7
8
9
finished
[None, None]
```

`Waiter` permet donc ici à `wait_job` d'attendre la fin de l'exécution de `count_up_to` avant de continuer. Il est possible de faire varier le temps de *sleep* pour constater qu'il ne s'agit pas d'un hasard : la première tâche se met en pause tant que la seconde n'a pas terminé son traitement.

`gather` est un utilitaire d'*asyncio* servant à exécuter «simultanément» (en concurrence) plusieurs tâches asynchrones dans la boucle événementielle. La fonction renvoie la liste des résultats des sous-tâches (le `[None, None]` que l'on voit dans la fin de l'exemple, nos tâches ne renvoyant rien).

D'autres utilisations de `Waiter` sont possibles, à des fins de synchronisation, par exemple pour gérer des verrous (*mutex*) entre plusieurs tâches.

3. Boucle d'or et les trois tâches

3.1. Une première boucle événementielle

Après avoir défini différentes tâches aysnchrones, il serait intéressant de construire le moteur pour les exécuter, la boucle événementielle. Cette boucle se charge de cadencer et d'avancer dans les tâches, tout en tenant compte des événements qui peuvent survenir.

Nous nous appuyions jusque-là sur la boucle fournie par *asyncio* (`asyncio.run`, `new_event_loop`) et sur son environnement (`sleep`, `gather`), mais il va nous être nécessaire de nous en détacher pour bien comprendre comment s'agencent les tâches, et donc de recoder ces outils.

Nous avons déjà un algorithme basique de boucle événementielle, que nous suivons pour le moment manuellement, pour traiter une tâche :

— Faire appel à `__await__` pour récupérer l'itérateur associé.
— Appeler continuellement `next` sur cet itérateur.
— S'arrêter quand une exception `StopIteration` est levée.

Il nous est donc possible d'écrire cela sous la forme d'une fonction `run_task` prenant une unique tâche en paramètre.

```
1 def run_task(task):
2     it = task.__await__()
3
4     while True:
5         try:
6             next(it)
7         except StopIteration:
8             break
```

Ce premier prototype de boucle fonctionne, nous pouvons l'utiliser pour exécuter l'une de nos tâches.

```
1 >>> run_task(complex_work())
2 Hello
3 World
```

Mais il est assez limité, ne traitant pas du tout la question de l'exécution concurrente ou du cadencement. Pour l'améliorer, nous créons donc la fonction `run_tasks`, recevant une liste de tâches. Les itérateurs de ces tâches seront placés dans une file (*FIFO*) par la boucle, qui pourra alors à chaque itération récupérer la prochaine tâche à traiter et la faire avancer d'un pas. Après quoi, si la tâche n'est pas terminée, elle sera ajoutée en fin de file pour être continuée plus tard.

```
def run_tasks(*tasks):
    tasks = [task.__await__() for task in tasks]

    while tasks:
        # On prend la première tâche disponible
        task = tasks.pop(0)
        try:
            next(task)
        except StopIteration:
            # La tâche est terminée
            pass
        else:
            # La tâche continue, on la remet en queue de liste
            tasks.append(task)
```

On obtient maintenant une exécution réellement concurrente. Le mécanisme de file (algorithme type *round-robin*) permet de traiter toutes les tâches de la même manière, sans en laisser sur le carreau. Ce sont néanmoins les tâches qui contrôlent la cadence, choisissant explicitement quand elles rendent la main à la boucle (`yield` / `await asyncio.sleep(0)` ou équivalents), lui permettant de passer à la tâche suivante.

Pour nous assurer du bon fonctionnement, on peut tester notre fonction avec nos coroutines `wait_job` et `count_up_to`.

```
>>> waiter = Waiter()
>>> run_tasks(wait_job(waiter), count_up_to(waiter, 10))
start
0
1
2
3
4
5
6
7
8
9
finished
```

3.2. Construire un environnement asynchrone

Cependant, un moteur asynchrone n'est rien sans les utilitaires qui vont avec. Nous avons vu la fonction `sleep` pour *asyncio* qui permet de patienter un certain nombre de secondes, et il serait utile d'en avoir un équivalent dans notre environnement.

Vous me direz que l'on utilise déjà `await asyncio.sleep(0)` dans nos coroutines et que ça ne pose pas de problème particulier, mais c'est justement parce que le paramètre vaut 0. Une autre valeur provoquerait une erreur parce que ne serait pas gérée par notre boucle événementielle.

Commençons par une tâche élémentaire toute simple, qui nous servira à construire le reste. Pour rendre la main à la boucle événementielle, il est nécessaire d'avoir un itérateur qui produit une valeur. Mais nous ne pouvons pas le faire directement depuis nos coroutines avec un `yield`, il faut nécessairement passer par une autre tâche que l'on `await`.

Il nous serait pratique d'avoir une tâche `interrupt`, où un `await interrupt()` serait équivalent à un `yield` / `await asyncio.sleep(0)`. C'est le cas avec la classe suivante.

```
class interrupt:
    def __await__(self):
        yield
```

La tâche est peu utile en elle-même, mais elle permet de construire autour d'elle un environnement de coroutines. Par exemple, on peut imaginer une coroutine qui rendrait la main à la boucle (et donc patienterait) tant qu'un temps (absolu) n'a pas été atteint.

```
import time

async def sleep_until(t):
    while time.time() < t:
        await interrupt()
```

Partant de là, une coroutine `sleep` se construit facilement en transformant une durée (temps relatif) en temps absolu.

```
async def sleep(duration):
    await sleep_until(time.time() + duration)
```

À titre d'exemple, voici une coroutine qui affiche des messages reçus en arguments, espacés par une certaine durée.

```
async def print_messages(*messages, sleep_time=1):
    for msg in messages:
```

```
        print(msg)
        await sleep(sleep_time)
```

On l'utilise ensuite avec `run_tasks` en instanciant deux coroutines pour bien voir que leurs messages s'intermêlent, et donc qu'il n'y a pas d'attente active : la boucle est capable de passer à la tâche suivante quand la première est bloquée, il lui suffit de rencontrer une interruption.

```
>>> run_tasks(
...     print_messages('foo', 'bar', 'baz'),
...     print_messages('aaa', 'bbb', 'ccc', sleep_time=0.7),
... )
foo
aaa
bbb
bar
ccc
baz
```

(Le résultat n'est pas très parlant ici vu qu'il manque de dynamisme, je vous invite à l'exécuter chez vous pour mieux vous en rendre compte.)

3.3. Interagir avec la boucle

La «boucle» que nous utilisons pour le moment ne permet aucune interaction : une fois lancée, il n'est par exemple plus possible d'ajouter de nouvelles tâches. Ça limite beaucoup les cas d'utilisation.

Pour remédier à cela, nous allons donc transformer notre fonction en classe afin de lui ajouter un état (la liste des tâches en cours) et une méthode `add_task` pour programmer de nouvelles tâches à la volée.

```
class Loop:
    def __init__(self):
        self.tasks = []

    def add_task(self, task):
        if hasattr(task, '__await__'):
            task = task.__await__()
        self.tasks.append(task)

    def run(self):
        while self.tasks:
            task = self.tasks.pop(0)
            try:
                next(task)
```

```
15              except StopIteration:
16                  pass
17              else:
18                  self.add_task(task)
```

Les deux premières lignes de la méthode `add_task` sont utiles pour reprogrammer une tâche déjà en cours (appel ligne 18), qui aura déjà été transformée en itérateur auparavant.

On peut aussi ajouter une méthode utilitaire, `run_task`, pour faciliter le lancement d'une tâche seule.

```
1  class Loop:
2      [...]
3
4      def run_task(self, task):
5          self.add_task(task)
6          self.run()
```

À l'utilisation, on retrouve le même comportement que précédemment.

```
1  >>> loop = Loop()
2  >>> loop.run_task(print_messages('foo', 'bar', 'baz'))
3  foo
4  bar
5  baz
```

Notre boucle possède maintenant un état, mais il n'est toujours pas possible d'interagir avec elle depuis nos tâches asynchrones, car nous n'avons aucun moyen de connaître la boucle en cours d'exécution.
Pour cela, nous ajoutons un attribut de classe `current` référençant la boucle en cous, réinitialisé à chaque `run`.

```
1  class Loop:
2      [...]
3
4      current = None
5
6      def run(self):
7          Loop.current = self
8          [...]
```

Dans un environnement réel, il nous faudrait réinitialiser `current` à chaque tour de boucle dans le `run`, pour permettre à plusieurs boucles de coexister. Mais le code proposé ici ne l'est qu'à titre d'exemple, on notera aussi que le traitement n'est pas *thread-safe*.

3.4. D'autres utilitaires asynchrones

Cet attribut `Loop.current` va nous être d'une grande utilité pour réaliser notre propre coroutine `gather`. Pour rappel, cet outil permet de lancer plusieurs coroutines «simultanément» et d'attendre qu'elles soient toutes terminées.

On peut commencer par reprendre notre classe `Waiter` pour étendre son comportement. Plutôt que de n'avoir qu'un état booléen, on le remplace par un compteur, décrémenté à chaque notification. On le dote alors d'une méthode `set` pour le notifier. L'attente d'un objet `Waiter` se termine une fois qu'il a été notifié `n` fois.

```
class Waiter:
    def __init__(self, n=1):
        self.i = n

    def set(self):
        self.i -= 1

    def __await__(self):
        while self.i > 0:
            yield
```

À partir de ce `Waiter` il devient très facile de recoder `gather`. Il suffit en effet d'instancier un `Waiter` en lui donnant le nombre de tâches, d'ajouter ces tâches à la boucle courante à l'aide de `Loop.current.add_task`, et d'attendre le `Waiter`.

Une petite subtilité seulement : les tâches devront être enrobées dans une nouvelle coroutine afin qu'elles notifient le `Waiter` en fin de traitement.

```
async def gather(*tasks):
    waiter = Waiter(len(tasks))

    async def task_wrapper(task):
        await task
        waiter.set()

    for t in tasks:
        Loop.current.add_task(task_wrapper(t))
    await waiter
```

On constate bien l'exécution concurrente des tâches, il est possible de faire varier le temps de pause pour observer les changements.

```
>>> loop = Loop()
>>> loop.run_task(
```

```
...     gather(
...         print_messages('foo', 'bar', 'baz'),
...         print_messages('aaa', 'bbb', 'ccc', sleep_time=0.7),
...     )
... )
foo
aaa
bbb
bar
ccc
baz
```

Et contrairement à notre précédent `run_tasks` qui permettait déjà celà, `gather` peut s'utiliser partout derrière un `await`, permettant de construire de vrais *workflows*.

```
>>> async def workflow():
...     await gather(
...         print_messages('a', 'b'),
...         print_messages('c', 'd', 'e'),
...     )
...     await print_messages('f', 'g')
...
>>> loop.run_task(workflow())
a
c
b
d
e
f
g
```

3.5. Utilitaires réseau (*sockets*)

Oublions ce `print_messages` et venons-en à des cas d'utilisation plus concrets. Les environnements asynchrones sont particulièrement adaptés aux programmes qui réalisent beaucoup d'opérations d'entrée/sortie (*I/O*), tels que des applications réseau. Voici par exemple comment nous pourrions intégrer des *sockets* (connecteurs réseau) à notre moteur asynchrone.

Nous utiliserons pour cela le type `socket` de Python, ainsi que la fonction `select` qui nous permet de savoir quand un fichier est prêt en lecture et/ou écriture. Le principe est alors, pour chaque opération, de vérifier si la *socket* est prête avant d'exécuter le traitement, et d'interrompre la coroutine le cas échéant afin de réessayer plus tard. À ce moment-là, la boucle événementielle reprend la main et peut continuer ses autres opérations pour ne pas les bloquer.

On construit une classe `AIOSocket`, reprenant l'interface de `socket`. Notre classe sera appelée avec une *socket* déjà instanciée, il ne reste alors plus qu'à instancier les sélecteurs pour la

surveiller en lecture et en écriture. Nous ajoutons les méthodes `close` et `fileno` pour respecter l'interface, ainsi que le protocole des gestionnaires de contexte.

```
import select

class AIOSocket:
    def __init__(self, socket):
        self.socket = socket
        self.pollin = select.epoll()
        self.pollin.register(self, select.EPOLLIN)
        self.pollout = select.epoll()
        self.pollout.register(self, select.EPOLLOUT)

    def close(self):
        self.socket.close()

    def fileno(self):
        return self.socket.fileno()

    def __enter__(self):
        return self

    def __exit__(self, *args):
        self.socket.close()
```

Et maintenant les coroutines de connexion, sur le modèle donné plus haut : attendre que la *socket* soit prête et exécuter l'opération ensuite.

```
class AIOSocket:
    [...]

    async def bind(self, addr):
        while not self.pollin.poll():
            await interrupt()
        self.socket.bind(addr)

    async def listen(self):
        while not self.pollin.poll():
            await interrupt()
        self.socket.listen()

    async def connect(self, addr):
        while not self.pollin.poll():
            await interrupt()
        self.socket.connect(addr)
```

Toujours sur ce modèle, on ajoute ensuite les coroutines de lecture/écriture. On notera juste que

la méthode `accept` d'une *socket* renvoie un couple *(socket, adresse)*. Nous ignorons ici l'adresse et emballons la *socket* dans une instance `AIOSocket`, afin de renvoyer un objet du même type.

```
class AIOSocket:
    [...]

    async def accept(self):
        while not self.pollin.poll(0):
            await interrupt()
        client, _ = self.socket.accept()
        return self.__class__(client)

    async def recv(self, bufsize):
        while not self.pollin.poll(0):
            await interrupt()
        return self.socket.recv(bufsize)

    async def send(self, bytes):
        while not self.pollout.poll(0):
            await interrupt()
        return self.socket.send(bytes)
```

Enfin, on ajoute un utilitaire pour créer une *socket* asynchrone de toutes pièces, reprenant les paramètres et valeurs par défaut de `socket`.

```
import socket

def aiosocket(family=socket.AF_INET, type=socket.SOCK_STREAM,
    proto=0, fileno=None):
    return AIOSocket(socket.socket(family, type, proto, fileno))
```

Avec ces *sockets* asynchrones, nous pouvons facilement créer des coroutines représentant un client et un serveur. Dans l'exemple qui suit, le serveur gère un unique client et ne fait que lui renvoyer le message reçu en l'inversant.

```
async def server_coro():
    with aiosocket() as server:
        await server.bind(('localhost', 8080))
        await server.listen()
        with await server.accept() as client:
            msg = await client.recv(1024)
            print('Received from client', msg)
            await client.send(msg[::-1])

async def client_coro():
```

```
    with aiosocket() as client:
        await client.connect(('localhost', 8080))
        await client.send(b'Hello World!')
        msg = await client.recv(1024)
        print('Received from server', msg)
```

```
>>> loop = Loop()
>>> loop.run_task(gather(server_coro(), client_coro()))
Received from client b'Hello World!'
Received from server b'!dlroW olleH'
```

On constate bien que rien n'est bloquant, les deux coroutines ont pu s'exécuter en concurrence, rendant la main à la boucle quand les *I/O* étaient indisponibles.

4. No Future

4.1. Mécanisme des *futures*

Le moteur asynchrone du chapitre précédent est assez peu efficace, notamment sa fonction `sleep`. En effet : la tâche est bien interrompue le temps de l'attente, mais elle est reprogrammée par la boucle à chaque itération, pour rien. De même pour la tâche `Waiter` qui n'a normalement pas besoin d'être programmée tant que son compteur ne vaut pas zéro.

On sait qu'une tâche est suspendue car elle attend qu'une condition (temporelle ou autre) soit vraie. il serait alors intéressant que la boucle événementielle ait connaissance de cela et ne cadence que les tâches dont les préconditions sont remplies.

Pour éviter ce problème, *asyncio* utilise un mécanisme de *futures*. Une *future* est une tâche asynchrone spécifique, qui permet d'attendre un résultat qui n'a pas encore été calculé. La *future* ne peut être relancée par la boucle événementielle qu'une fois ce résultat obtenu.

Il se trouve que le `yield` utilisé dans nos tâches pour rendre la main à la boucle peut s'accompagner d'une valeur, comme dans tout générateur. Ici, il va nous servir à communiquer avec la boucle, pour lui indiquer la *future* en cours. C'est ce que fait `asyncio.sleep` avec une durée non nulle par exemple.

On peut commencer avec un prototype de *future* tout simple, sur le modèle de notre première classe `Waiter`.

```
class Future:
    def __await__(self):
        yield self
        assert self.done
```

Nous n'avons pas besoin de boucle ici, puisque la tâche ne devrait pas être programmée plus de deux fois : une première fois pour démarrer l'attente, et une seconde après que la condition soit remplie pour reprendre le travail de la tâche appelante. On place néanmoins un `assert` pour s'assurer que ce soit bien le cas.

Lorsque, depuis une coroutine, on fera un `await Future()`, la valeur passée au `yield` remontera le flux des appels jusqu'à la boucle événementielle, qui la recevra en valeur de retour de `next`. Ainsi, un `yield self` depuis la classe `Future` permettra à la boucle d'avoir accès à la *future* courante. C'est le seul moyen pour la boucle d'en avoir une référence, puisqu'elle ne connaît sinon que la tâche asynchrone englobante.

Pour améliorer notre classe `Future`, on va l'agrémenter d'une méthode `set` afin de signaler que le traitement est terminé. En plus de cela, la méthode se chargera aussi de reprogrammer notre tâche au niveau de la boucle événementielle (c'est à dire de l'ajouter à nouveau aux tâches à exécuter, afin qu'elle soit prise en compte à l'itération suivante).

Pour connaître la tâche à cadencer, on va utiliser l'attribut `task` de l'objet `Future`. Il n'existe pas encore pour le moment, mais sa valeur lui sera attribuée par la boucle événementielle lorsque la tâche sera interrompue.

```
class Future:
    def __init__(self):
        self._done = False
        self.task = None

    def __await__(self):
        yield self
        assert self._done

    def set(self):
        self._done = True
        if self.task is not None:
            Loop.current.add_task(self.task)
```

4.2. Intégration à la boucle événementielle

Notre tâche `Future` est maintenant complète, mais le reste du travail est à appliquer du côté de la boucle, pour qu'elle les traite correctement.

— Premièrement, il faut que quand une tâche s'interrompt sur une *future*, la boucle définisse l'attribut `task` de la *future* comme convenu.
— Ensuite, la boucle ne doit pas reprogrammer une telle tâche, puisque ça provoquerait un doublon lorsque la *future* sera notifiée.
— Enfin, il est nécessaire de lier les *futures* à des événements, pour que l'appel à `set` et donc le déclenchement de la tâche soient automatiques.

On commence par les deux premiers points, faciles à ajouter à la méthode `run` de `Loop`.

```python
class Loop:
    [...]

    def run(self):
        Loop.current = self
        while self.tasks:
            task = self.tasks.pop(0)
            try:
                result = next(task)
            except StopIteration:
                continue

            if isinstance(result, Future):
                result.task = task
            else:
                self.tasks.append(task)
```

4.3. Événements temporels

Pour le troisième point, on va formaliser l'idée d'événements. Les plus simples à mettre en place sont les événements temporels, et ce sont donc les seuls que nous allons traiter ici. En effet, la boucle a conscience du temps qui s'écoule et peut déclencher des actions en fonction de ça. Le but sera donc d'associer un temps à une *future*, et d'y faire appel dans la boucle.

Tout d'abord, on crée une classe `TimeEvent` associant ces deux éléments. On rend les objets de cette classe ordonnables, en implémentant les méthodes spéciales `__eq__` (opérateur `==`) et `__lt__` (opérateur `>`) puis en appliquant le décorateur `functools.total_ordering` pour générer les méthodes des autres opérateurs d'ordre.
On a besoin que les objets soient ordonnables pour trouver facilement les prochains événements à déclencher.

```python
from functools import total_ordering

@total_ordering
class TimeEvent:
    def __init__(self, t, future):
        self.t = t
        self.future = future

    def __eq__(self, rhs):
        return self.t == rhs.t

    def __lt__(self, rhs):
        return self.t < rhs
```

On intègre les événements temporels à notre boucle en la dotant d'une méthode `call_later`. Cette méthode reçoit un temps (absolu) et une *future*, les associe dans un objet `TimeEvent` qu'elle ajoute à la file des événements. On utilise pour la file un objet `heapq` qui permet de conserver un ensemble ordonné : le premier événement de la file sera toujours le prochain à exécuter.

`heapq` est un module fournissant des fonctions (`heappush`, `heappop`) qui s'appuient sur une liste (`self.handlers` dans le code qui suit) pour garder une file cohérente.

```
import heapq

class Loop:
    [...]

    def __init__(self):
        self.tasks = []
        self.handlers = []

    def call_later(self, t, future):
        heapq.heappush(self.handlers, TimeEvent(t, future))
```

Dans le cœur de la boucle (méthode `run`), il suffit alors de regarder l'événement en tête de file, et de le déclencher si besoin (si son temps est atteint). Déclencher l'événement signifie notifier la *future* qui lui est associée (appeler sa méthode `set`). L'effet sera donc immédiat, la *future* ajoutera la tâche suspendue aux tâches courantes, et celle-ci sera prise en compte par la boucle pendant l'itération. Le reste de la méthode `run` reste inchangé.

```
class Loop:
    [...]

    def run(self):
        Loop.current = self
        while self.tasks or self.handlers:
            if self.handlers and self.handlers[0].t <= time.time():
                handler = heapq.heappop(self.handlers)
                handler.future.set()

            if not self.tasks:
                continue
            task = self.tasks.pop(0)
            try:
                result = next(task)
            except StopIteration:
                continue

            if isinstance(result, Future):
                result.task = task
```

```
21              else:
22                  self.tasks.append(task)
```

4.4. Utilisation des *futures*

Notre boucle gérant correctement les événements temporels, on peut maintenant réécrire `sleep` avec une *future* et un *time-handler*. Tout ce qu'a à faire `sleep` c'est de convertir une durée en temps absolu, instancier une *future* et l'ajouter à la boucle en appelant `call_later`.

```
1  import time
2
3  async def sleep(t):
4      future = Future()
5      Loop.current.call_later(time.time() + t, future)
6      await future
```

Il suffit qu'une coroutine exécute `await sleep(...)` pour que tout le mécanisme se mette en place :

— Une *future* est instanciée, un événement temporel lui est associé dans la boucle, réglé sur la durée demandée.
— La coroutine est retirée de la liste des tâches à traiter.
— La boucle continue son travail, en itérant sur les autres tâches, jusqu'à ce que l'événement temporel se produise.
— Là, elle déclenche la notification de la *future*, la coroutine est donc rajoutée à la liste des tâches.
— La boucle reprend alors l'exécution de la coroutine précédemment suspendue.

```
1  >>> async def foo():
2  ...     print('before')
3  ...     await sleep(5)
4  ...     print('after')
5  ...
6  >>> loop = Loop()
7  >>> loop.run_task(foo())
8  before
9  after
```

Notre boucle possède encore bien des défauts, comme celui de faire de l'attente active (bloquer le processeur) quand il n'y a rien à exécuter. L'implémentation d'`asyncio` est bien sûr plus évoluée que ce qui est présenté ici.

5. Et pour quelques outils de plus

`async def` et `await` ne sont pas les seuls mots-clés introduits par la version 3.5 de Python. Deux nouveaux blocs ont aussi été ajoutés : les boucles asynchrones (`async for`) et les gestionnaires de contexte asynchrones (`async with`).

Ils sont similaires à leurs équivalents synchrones mais utilisent des méthodes spéciales qui font appel à des coroutines. Et ils ne sont utilisables qu'au sein de coroutines (de la même manière qu'`await`).

Aussi, Python n'a pas arrêté d'évoluer après cette version 3.5, et de nouveaux outils pour la programmation asynchrones sont venus s'y ajouter depuis.

5.1. Itérables & générateurs asynchrones

5.1.1. Itérables asynchrones

Pour rappel, un itérable est un objet possédant une méthode `__iter__` renvoyant un itérateur. Et un itérateur est un objet possédant une méthode `__next__` qui renvoie le prochain élément à chaque appel. Plus d'informations à ce sujet ici .

Sur ce même modèle, un itérable asynchrone est un objet doté d'une méthode `__aiter__` qui renvoie un itérateur asynchrone (`__aiter__` est une méthode synchrone).
Et un itérateur asynchrone possède une méthode-coroutine `__anext__`, renvoyant le prochain élément et pouvant user de tous les outils asynchrones.

Un itérateur synchrone se termine quand sa méthode `__next__` lève une exception `StopIteration`. Dans le cas des itérateurs asynchrones, c'est une exception `AsyncStopIteration` qui sera levée.

La boucle `async for` parcourant l'itérateur sera suspendue pendant les attentes (rendant la main à la boucle événementielle).

Le code qui suit présente la classe `ARange`, un itérable asynchrone qui produit des nombres à la manière de `range`, mais en se synchronisant sur un événement extérieur (ici un `sleep(1)`). `ARange` représente l'itérable et `ArangeIterator` l'itérateur associé (qui n'a jamais besoin d'être utilisé directement). `ARange` en elle-même n'a rien d'asynchrone, tout le code asynchrone se trouve dans la classe de l'itérateur.

```
1  class ARange:
2      def __init__(self, stop):
3          self.stop = stop
4
5      def __aiter__(self):
6          return ARangeIterator(self)
7
8
9  class ARangeIterator:
10     def __init__(self, arange):
```

```
        self.arange = arange
        self.i = 0

    async def __anext__(self):
        if self.i >= self.arange.stop:
            raise StopAsyncIteration
        await sleep(1)
        i = self.i
        self.i += 1
        return i
```

Pour tester l'itérable dans notre environnement, définissons une simple coroutine utilisant un `async for` :

```
>>> async def test_for():
...     async for val in ARange(5):
...         print(val)
...
>>> loop = Loop()
>>> loop.run_task(test_for())
0
1
2
3
4
```

5.1.2. Générateurs asynchrones

Les choses se simplifient en Python 3.6 où il devient possible de définir des générateurs asynchrones. Il suffit d'un `yield` utilisé dans une coroutine pour la transformer en fonction génératrice asynchrone.

```
async def arange(stop):
    for i in range(stop):
        await sleep(1)
        yield i
```

`arange` s'utilise exactement de la même manière que la classe `ARange` précédente (remplacez `ARange(5)` par `arange(5)` dans l'exemple plus haut pour le vérifier), mais avec un code bien plus court.

En Python 3.6 la syntaxe `async for` devient aussi utilisable dans les listes / générateurs / ensembles / dictionnaires en intension, toujours depuis une coroutine.

```
>>> async def test_for():
...     print([x async for x in arange(5)])
...
>>> loop = Loop()
>>> loop.run_task(test_for())
[0, 1, 2, 3, 4]
```

5.2. Gestionnaires de contexte asynchrones

Un gestionnaire de contexte est défini par ses méthodes `__enter__` et `__exit__` permettant d'exécuter du code en entrée et en sortie de bloc `with` (voir ici).

Le gestionnaire de contexte asynchrone est défini sur le même modèle, avec des coroutines `__aenter__` et `__aexit__`. Elles sont donc exécutées respectivement à l'entrée et à la sortie du bloc `async with`, utilisé dans une coroutine.

Par exemple, voici un gestionnaire de contexte permettant de créer un serveur autour de l'objet `aiosocket` que nous avons utilisé au chapitre 3. La coroutine `__aenter__` se charge de démarrer le serveur (*bind* et *listen*), et `__aexit__` le clôt (*close*).

```
class Server:
    def __init__(self, addr):
        self.socket = aiosocket()
        self.addr = addr

    async def __aenter__(self):
        await self.socket.bind(self.addr)
        await self.socket.listen()
        return self.socket

    async def __aexit__(self, *args):
        self.socket.close()
```

Encore une fois, nous définissons une coroutine pour pouvoir tester notre objet. On reprend le même principe que précédemment d'un serveur qui renvoie juste l'inverse du message reçu. Et l'on réutilise `client_coro` pour jouer le rôle du client.

```
>>> async def test_with():
...     async with Server(('localhost', 8080)) as server:
...         with await server.accept() as client:
...             msg = await client.recv(1024)
...             print('Received from client', msg)
...             await client.send(msg[::-1])
```

```
...
>>> loop = Loop()
>>> loop.run_task(gather(test_with(), client_coro()))
Received from client b'Hello World!'
Received from server b'!dlroW olleH'
```

Cette fois-ci c'est Python 3.7 qui est venu simplifier les choses, en ajoutant le support des gestionnaires de contexte asynchrones à la `contextlib`. Ainsi, il devient possible d'utiliser un décorateur `asynccontextmanager` pour transformer un générateur asynchrone en gestionnaire de contexte asynchrone. L'instruction `yield` permet de séparer le code d'initialisation de celui de fermeture du bloc `async with`.

```
from contextlib import asynccontextmanager

@asynccontextmanager
async def server(addr):
    socket = aiosocket()
    try:
        await socket.bind(addr)
        await socket.listen()
        yield socket
    finally:
        socket.close()
```

`server` s'utilise de la même manière que la classe `Server` précédente.

Bien sûr, tout ce qui est présenté dans cet article ne l'est qu'à titre d'exemple. Le but est d'étudier comment fonctionne dans les grandes lignes un moteur asynchrone, et en particulier *asyncio*, mais pas de le remplacer. Si vous êtes intéressé par une alternative à *asyncio*, jetez un œil du côté de *trio* qui utilise une approche différente.

L'objet de cet article a été présenté à Bordeaux lors de la PyConFr 2019, dont vous pouvez retrouver les diapositives sur Github et la conférence sur Youtube :

Élément externe (Video) —
Consultez cet élément à l'adresse `https://www.youtube.com/embed/WdQmNl2kShI?feature=oembed`.