Zeste de savoir

La puissance cachée des coroutines

12 août 2019

Table des matières

Les coroutines sont un mécanisme implémenté dans certains langages de programmation pour rendre possible l'exécution concurrente de parties indépendantes d'un programme. Relativement populaires dans les années 60, elles ont peu à peu été délaissées au profit des *threads* qui sont aujourd'hui supportés par la grande majorité des systèmes d'exploitation multitâches.

Et puis une quarantaine d'années plus tard, lors de la conférence PyCon' 2009, David Beazley, Pythoniste émérite, auteur de plusieurs livres et créateur de la bibliothèque `PLY` (Python Lex-Yacc) a présenté un long tutoriel à propos de l'implémentation des coroutines en Python, en essayant de montrer leur puissance. En particulier, il a montré la façon dont elles pouvaient être utilisées pour créer des boucles événementielles permettant de gérer des entrées-sorties asynchrones *sans utiliser de threads*. Néanmoins, encore à cette époque, les coroutines étaient considérées au mieux comme une curiosité un peu magique et pas très intuitive.

Les années passant, la problématique des IO asynchrones a été de plus en plus discutée dans le monde de Python, jusqu'à l'introduction, par Guido van Rossum en personne, de `asyncio` dans la bibliothèque standard de la version 3.4 de Python. Cette bibliothèque, dont le but est de donner un socle commun à d'autres frameworks événementiels reconnus et populaires (comme Twisted), repose sur une utilisation aussi astucieuse qu'intensive… des **coroutines**.

Inutile de préciser qu'il s'agit d'un signe avant-coureur : la programmation par coroutines a de grandes chances de devenir populaire dans les années à venir et les futures versions de Python !

Cet article a pour but de vous faire découvrir la programmation de coroutines en Python. Nous l'illustrerons en développant deux exemples de programmes aux applications réalistes, inspirés de la présentation initiale de David Beazley.

Icône : *"Fire Black Magic"* par firefrank

1. Ce que cache le mot-clé yield

S'il y a un mot-clé que les Pythonistes utilisent couramment en s'imaginant connaître son fonctionnement par coeur, c'est bien `yield`. Cette petite instruction, en apparence très simple, permet de créer en quelques coups de cuiller à pot des *générateurs* sur lesquels il est facile d'itérer :

```
>>> def menu():
...     yield "spam"
...     yield "eggs"
...     yield "bacon"
...
>>> for elt in menu():
...     print(elt)
...
spam
eggs
bacon
```

Son fonctionnement a ceci de remarquable que l'exécution de la fonction est suspendue entre deux itérations. Remarquez que l'appel de la fonction `count()` dans l'exemple suivant ne fait absolument pas broncher Python, alors qu'il s'agit pourtant d'une boucle infinie !

```
>>> def count():
...     i = 0
...     while True:
...         print("tick")
...         i += 1
...         yield i
...
>>> gen = count()
>>> next(gen)
tick
1
>>> next(gen)
tick
2
>>> next(gen)
tick
3
```

On pourrait se dire qu'avec ces deux exemples, nous avons fait le tour de la question, et ce serait la fin de cet article… Mais saviez-vous que l'on pouvait également **envoyer des objets** à un générateur en cours d'exécution ?

Si ce n'est pas le cas et que vous pensiez vraiment bien connaître Python, accrochez-vous à quelquechose de solide, car votre représentation de l'univers risque de s'effondrer à la vue du prochain exemple :

```
>>> def receiver():
...     print("ready!")
...     while True:
...         data = yield
...         print("received {!r}".format(data))
...
>>> gen = receiver()
>>> next(gen)
ready!
>>> gen.send("Hello, World!")
received 'Hello, World!'
>>> gen.send(['spam', 'eggs', 'bacon'])
received ['spam', 'eggs', 'bacon']
```

… Vous êtes encore là ?

Cette fonction `receiver()`, qui reçoit des données grâce au mot-clé `yield`, prouve que les générateurs de Python sont bien plus que ce qu'ils semblent être à première vue. Il ne s'agit pas uniquement de *producteurs* de données que l'on peut appeler ou interrompre quand bon nous semble : on peut également s'en servir comme *consommateurs*.

En somme, ce mot-clé `yield` n'est pas seulement un sucre syntaxique permettant de créer des objets itérables avec une simple fonction, mais bel et bien un **mécanisme de coroutines**, c'est-à-dire un point de suspension dans l'exécution d'une fonction, que l'on peut utiliser pour échanger des données avec elle.

Dans la suite de cet article, nous allons évoquer deux *patterns* de programmation "utilisables dans la vraie vie" avec des coroutines.

2. Y'a du monde dans le pipe !

L'exemple le plus populaire pour illustrer la programmation avec des coroutines est certainement la simulation des *pipes* des systèmes Unix.

Si vous avez déjà utilisé un shell, vous connaissez sûrement cette syntaxe utilisant le symbole `|` qui sert à chaîner divers traitements les uns derrière les autres. Par exemple sachant que la commande `ps ax` sert à afficher des informations sur les processus en cours d'exécution et que `grep python` permet de filtrer toutes les lignes d'un flux de texte qui ne contient pas le mot `"python"`, on peut les assembler pour obtenir des informations sur toutes les instances de python qui sont en train de tourner sur le système, comme ceci :

```
$ ps ax | grep python
 1431 ?        S      6:48 /usr/bin/python -O
     /usr/share/wicd/daemon/wicd-daemon.py
 1498 ?        S      2:30 /usr/bin/python -O
     /usr/share/wicd/daemon/monitor.py
 2667 ?        Sl     1:30 /usr/bin/python -O
     /usr/share/wicd/gtk/wicd-client.py --tray
17662 pts/19   S+     0:01 python3
31348 ?        SNl    0:20 /usr/bin/python3 /usr/bin/update-manager
```

La façon dont ce *pipeline* fonctionne est assez intéressante : les programmes `ps` et `grep`, dans notre exemple, sont exécutés **simultanément**. La sortie de `ps` est simplement envoyée dans l'entrée standard de `grep`. Ainsi, `ps` produit des données, `grep` les consomme et produit à son tour une sortie... que l'on pourrait elle-même rediriger vers un autre élément.

Par exemple, on peut rajouter un deuxième `grep` derrière le premier pour ne filtrer que les lignes correspondant à des programmes connectés à un terminal, en filtrant sur le mot `'pts'` :

```
$ ps ax | grep python | grep pts
17662 pts/19   S+     0:01 python3
```

Puissant, n'est-ce pas ? Eh bien on peut tout à fait obtenir la même souplesse de traitement avec des coroutines en Python !

Commençons par le plus facile : pour afficher les données qui transitent dans le pipeline, nous allons écrire un `printer` qui se contente d'afficher tout ce qu'il voit passer :

```
def printer(prefix=''):
    while True:
        data = yield
        print('{}{}'.format(prefix, data))
```

L'argument optionnel `prefix` pourra vous servir plus tard pour déterminer *quel* printer est en train d'afficher des choses à l'écran.

Essayons-le dans la console :

```
>>> ptr = printer()
>>> ptr.send('plop')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't send non-None value to a just-started generator
```

Ça commence mal. Python nous explique que notre printer ne peut pas recevoir de données *à froid* dès le départ. En fait lorsque l'on crée un générateur, la fonction n'est pas démarrée tout de suite : il faut itérer une fois dessus pour entrer dans la boucle.

```
>>> ptr = printer()   # création du générateur
>>> next(ptr)         # démarrage de la coroutine
>>> ptr.send('plop')  # on peut maintenant lui envoyer des données
plop
```

Vu que ce n'est pas très pratique, et que l'on risque d'oublier souvent cette première itération, on va créer un petit décorateur `@coroutine` qui se chargera de démarrer les générateurs pour nous lors de leur instanciantion.

```
def coroutine(func):
    def starter(*args, **kwargs):
        gen = func(*args, **kwargs)
        next(gen)
        return gen
    return starter

@coroutine
def printer(prefix=''):
    while True:
        data = yield
        print('{}{}'.format(prefix, data))
```

Essayons :

```
>>> ptr = printer()
>>> ptr.send('spam')
spam
```

Voilà qui est mieux ! Créons maintenant un autre élément du pipeline : le *grepper*. Celui-ci va prendre en argument un motif (`pattern`) à rechercher dans les données qu'on lui envoie, et une cible (`target`) à qui faire suivre la donnée si celle-ci contient le motif :

```
@coroutine
def grepper(pattern, target):
    while True:
        data = yield
        if pattern in data:
            target.send(data)
```

2. Y'a du monde dans le pipe !

Essayons de mettre nos blocs bout à bout :

```
>>> ptr = printer()
>>> gpr = grepper(pattern='python', target=ptr)
>>> gpr.send("python c'est coolympique !")
python c'est coolympique !
>>> gpr.send("ruby c'est pas mal non plus.")
>>> gpr.send("java n'en parlons pas !")
>>> gpr.send("mais python c'est le top.")
mais python c'est le top.
```

Maintenant, on va essayer autre chose en créant des routes parallèles dans notre *pipeline*. Pour cela, nous avons besoin d'un *multiplexer* qui va nous permettre de relayer la même donnée vers plusieurs cibles en même temps :

```
@coroutine
def multiplexer(*targets):
    while True:
        data = yield
        for tgt in targets:
            tgt.send(data)
```

Son code est très simple, là encore. Pour tester ce multiplexer, créons le pipeline illustré sur la figure suivante, qui nous servira à filtrer toutes les lignes contenant "python" ou "ruby" :

http://zestedesavoir.com/media/galleries/1748/

```
>>> ptr = printer()
>>> mpxr = multiplexer(grepper('python', ptr), grepper('ruby',
    ptr))
>>> mpxr.send("python c'est coolympique !")
python c'est coolympique !
>>> mpxr.send("ruby c'est pas mal non plus.")
ruby c'est pas mal non plus.
>>> mpxr.send("java n'en parlons pas !")
>>>
```

Vous aurez compris le principe, je pense : nous avons trois petites fonctions de quelques lignes que nous pouvons connecter entre elles pour créer des arrangements complexes de traitements. Vous pouvez vous amuser à rajouter des noeuds dans le pipeline (par exemple pour mettre les

données en majuscules, ou les numéroter…) et leur faire manger des giga-octets de texte, ligne par ligne. Les possibilités sont infinies.

Si le sujet vous intéresse, vous pouvez essayer de construire des arbres de traitement en fonction d'un fichier de configuration. En ce qui nous concerne, nous allons passer au niveau supérieur avec une application un peu moins triviale.

3. Un parseur (très) léger et (plutôt) performant

Vous aurez compris que les coroutines sont pratiques pour simplifier le développement de fonctions *incrémentales*, auxquelles on envoie des données petit à petit, au fur et à mesure qu'elles arrivent. Par ailleurs, nous avons vu plus haut un exemple dans lequel, au moyen de petites fonction atomiques, nous avons créé un traitement qui se représente sous la forme d'un graphe arbitrairement complexe.

Il y a des programmes qu'on apprécie d'autant plus quand ils traitent les données de façon incrémentale, et dont la structure prend généralement la forme d'un graphe, ou plutôt d'une machine à états : **les parseurs**.

Dans son intervention au PyCon de 2009 , David Beazley présentait un cas d'utilisation intéressant, dans lequel il se sert de coroutines pour parser un document XML de façon incrémentale. Cependant, à l'époque, la syntaxe de Python, plus limitée qu'aujourd'hui, rendait son exemple un peu laborieux, car elle n'incluait pas la construction `yield from` qui a été introduite dans Python 3.3. Nous allons ici nous inspirer de son exemple, mais pour arriver à un résultat beaucoup plus simple et élégant que ce qu'il était possible de faire avant l'arrivée de cette construction dans le langage.

Dans toute la suite de cet article, nous allons implémenter un programme `xmlgrep` qui va parser, de façon incrémentale, un fichier xml pour n'afficher sur la sortie standard que le contenu des noeuds *dont on lui aura passé le chemin*. Par exemple, imaginons que d'une façon ou d'une autre, nous voulions récupérer les titres des derniers articles de zestedesavoir.com sur le flux rss correspondant . Nous pourrions nous servir de `xmlgrep` de cette façon :

```
$ wget -O - https://zestedesavoir.com/articles/flux/rss/ |
    ./xmlgrep.py item/title
[item/title] Interview : Rencontre avec Taguan
[item/title] Programmez en langage d'assemblage sous Linux !
[item/title] Zeste de Savoir passe en version 1.7
[item/title] Assemblée générale de l'association Zeste de Savoir
[item/title] La géographie, une approche pour comprendre le monde
    qui nous entoure
```

L'avantage d'un parseur qui travaille de façon incrémentale est qu'ici, que nous ayons quelques kilo-octets ou plusieurs centaines de giga-octets de données XML à parser, *le programme ne prendra pas plus de place en mémoire*, et qu'il produira des résultats sur la sortie standard au fur et à mesure qu'il consommera les données correspondantes.

Mais d'abord, rappelons à quoi sert ce `yield from` dont nous avons parlé.

3.1. La syntaxe yield from de Python 3

Prenons le générateur que voici :

```
def gen():
    yield 'chiffres de 1 à 3'
    for x in range(1, 4):
        yield x
```

Celui-ci se contente, après avoir produit une première donnée, d'envoyer les éléments générés par la fonction `range(1, 4)` :

```
>>> for x in gen(): print(x)
...
chiffres de 1 à 3
1
2
3
```

La syntaxe `yield from` permet en fait de dire à la fonction appelante « *ce générateur va me remplacer temporairement* ». Ainsi, au lieu de boucler explicitement dessus, on peut **remplacer** notre générateur par `range(1, 4)` comme ceci :

```
def gen():
    yield 'chiffres de 1 à 3'
    yield from range(1, 4)
```

Même si cela a juste l'air d'un simple raccourcis, ce n'est absolument pas le cas : lorsque qu'une coroutine utilise `yield from`, **les données que l'on envoie à cette coroutine seront automatiquement transmises à celle appelée par le `yield from`**. Ça a l'air d'un détail à première vue, mais grâce à cela, on est en mesure de *passer dynamiquement le relais à une autre coroutine* en fonction des données reçues. Il ne nous en faut pas plus pour implémenter une véritable machine à états.

3.2. Un prototype pour tout mettre en place

Pour commencer, il faut savoir que de nombreux parseurs XML en Python (parmi lesquels celui de la bibliothèque standard), sont capables de générer un *flux d'événements* au fur et à mesure qu'ils consomment des données.

C'est exactement le rôle, par exemple, de la fonction `iterparse()` du module standard `xml.etree.ElementTree` . Chacun de ces événements est associé à un élément de l'arbre XML en cours de construction. Nous en trouverons deux types :

— `start` : une balise ouvrante vient d'être rencontrée. L'élément associé contient au moins tous les attributs de la balise ;
— `end` : une balise fermante vient d'être rencontrée. L'élément associé contient les données de la balise.

```
>>> from urllib.request import urlopen
>>> from xml.etree.ElementTree import iterparse
>>> res = urlopen('https://zestedesavoir.com/articles/flux/rss/')
>>> gen = iterparse(res, ('start', 'end'))
>>> next(gen)
('start', <Element 'rss' at 0x7ff7890c2688>)
>>> next(gen)
('start', <Element 'channel' at 0x7ff7890d33b8>)
>>> next(gen)
('start', <Element 'title' at 0x7ff7890d3408>)
>>> next(gen)
('end', <Element 'title' at 0x7ff7890d3408>)
```

Notre parseur va recevoir ce flux d'événements, et réagir de façon à afficher tout le contenu des balises `<title>` contenues à l'intérieur d'une balise `<item>`.

```
<rss>
    <channel>
        <title>Ce titre doit être ignoré</title>
        ...
        <item>
            <title>Ce titre sera extrait</title>
            ...
        </item>
        <item>
            <title>Celui-ci aussi</title>
            ...
        </item>
        ...
    </channel>
</rss>
```

Nous allons commencer par coder ces fonctions "en dur" avant de trouver une façon de créer dynamiquement ces coroutines. Commençons par une fonction `title()`. Cette fonction est chargée d'afficher le contenu du prochain élément `<title>` qu'elle verra passer. Pour ce faire, elle va réagir à l'événement `(end, title)` :

```
def title():
    while True:
        event, element = yield
```

```
        tag = element.tag
        if (event, tag) == ('end', 'title'):
            print(element.text)
            return
```

Cependant, nous ne voulons pas afficher n'importe quels éléments `<title>`, mais seulement ceux qui sont contenus à l'intérieur d'une balise `<item>`. Pour cette raison, nous allons créer une coroutine `item()`, dont le but sera d'écouter tous les événements compris entre un `(start, item)` et un `(end, item)`, pour passer le relai à la fonction `title()` lorsque l'on entrera dans une balise `<title>` :

```
def item():
    while True:
        event, element = yield
        tag = element.tag
        if (event, tag) == ('start', 'title'):
            yield from title()
        elif (event, tag) == ('end', 'item'):
            return
```

Conceptuellement, c'est plutôt simple, non ?

Ajoutons une dernière fonction qui va écouter tous les événements qui passent pour détecter le moment où l'on entrera dans une balise `<item>`, et passer ce faisant le contrôle à la coroutine `item()`.

```
def root():
    while True:
        event, element = yield
        tag = element.tag
        if (event, tag) == ('start', 'item'):
            yield from item()
```

Et... c'est tout !

Il ne nous reste plus qu'à rajouter un peu de code pour initialiser la machine à états et supprimer les éléments produits par le parseur au fur et à mesure quand on n'en a plus besoin pour libérer la mémoire.

```
state_machine = root()
next(state_machine)

rss = urlopen('https://zestedesavoir.com/articles/flux/rss')
for event, element in ElementTree.iterparse(rss, ('start', 'end')):
```

```
    state_machine.send((event, element))
    if event == 'end':
        element.clear()
```

Et voilà, nous avons un premier parseur tout à fait fonctionnel ! Cela dit, ce code n'est pas très souple. Ce serait quand même mieux si on pouvait le factoriser de manière à créer des parseurs adaptés à n'importe quelle recherche, pour qu'on puisse l'utiliser comme je vous l'ai montré plus haut.

3.3. Génération automatique

Pour comprendre comment nous pouvons généraliser ce code, remarquons que notre parseur se comporte exactement comme une machine à états :

D'ailleurs, pour bien comprendre la puissance de cette structure, imaginons que nous voulions maintenant parser tous les noeuds `item/description` en plus de `item/title`.

Comme vous le voyez, nos fonctions `root()`, `item()` et `title()`, dans l'exemple précédent, représentent des *états* du parseur, alors que les transitions sont réalisées par nos instructions `yield from` et `return`.

En toute rigueur, cette machine à états souffre d'un léger défaut : si elle fait parfaitement son travail dans notre exemple actuel, elle risque en revanche de provoquer un comportement contre-intuitif dans le cas de structures xml récursives comme celle-ci :

```
<rss>
    <item>
        <title>Ce titre sera extrait</title>
        <item>
            <foo>
                <title>Celui-ci aussi</title>
            </foo>
```

```
        </item>
        <title>Mais pas celui-là</title>
    </item>
</rss>
```

Néanmoins, nous ne nous pencherons pas sur ce problème dans le cadre de cet article pour garder le propos relativement simple. Une piste d'amélioration sera abordée plus bas.

Nous pourrions configurer cette machine à états dans la structure suivante :

```
STATES = {
    '__start__': {
        'transitions': {
            'item': 'item',
        },
    },
    'item': {
        'transitions': {
            'title': 'item/title',
            'description': 'item/description',
        },
    },
    'item/title': {
        'transitions': {},
        'do_print': True
    },
    'item/description': {
        'transitions': {},
        'do_print': True
    }
}
```

Pour chaque état, on crée une transition vers un nouvel état lorsque l'on veut qu'il réagisse à un évènement `start` particulier. Chaque état retournera à l'état précédent lorsqu'il croisera son évènement `end`. Enfin, si dans un état donné, le `end` doit également déclencher l'affichage du contenu de la balise au moment de sortir, on rajoute un argument `do_print` que l'on positionne sur `True`

Le plus gros du travail sera donc de créer une fonction `state()` générique, qui tire partie de cette structure.

```
def state(tag_name, transitions={}, do_print=False, path=''):
    while True:
        event, element = yield
        tag = element.tag
        if event == 'start':
```

```
            # S'il existe une transition sortante sur la balise qui vient de
            # démarrer, on passe le relai à l'état pointé par la transition
            if tag in transitions:
                args = STATES[transitions[tag]]
                yield from state(tag, **args)
        elif tag == tag_name:
            # Si on croise la balise fermante de l'état courant,
            # on repasse la main à l'état précédent, en affichant
            # éventuellement le contenu de la balise
            if do_print:
                print('[{}] {}'.format(path, element.text))
            return
```

Avouez qu'on aurait pu s'attendre à plus complexe. Remarquez que l'on gagne du temps en passant à la fonction le contenu exact de l'état modélisé dans le dictionnaire, au travers de ses arguments optionnels avec la syntaxe `**args`… Devant la dose de magie noire que nous avons employée jusqu'à maintenant, je pense que l'on n'est plus à ceci près.

Il ne nous reste plus qu'à générer le contenu du dictionnaire `STATES` à partir d'un chemin du style `"item/title"`, ce qui demande un peu de réflexion :

```
STATES = {'__start__': {'transitions': {}, 'path': '/'}}

def add_path(path):
    nodes = deque(path.strip('/').split('/'))
    current_state = STATES['__start__']
    current_path = []
    while nodes:
        tag = nodes.popleft()
        current_path.append(tag)
        next_state = current_state['transitions'].get(tag, None)
        if next_state:
            current_state = STATES[next_state]
            continue
        path = '/'.join(current_path)
        current_state['transitions'][tag] = path
        current_state = {'transitions': {}, 'path': path}
        STATES[path] = current_state
    current_state['do_print'] = True
```

Et voilà ! Terminons ce script en lisant les données directement depuis l'entrée standard du programme et nous avons maintenant un parseur utilisable.

```
$ export URL=https://zestedesavoir.com/articles/flux/rss/
$ wget -O - $URL 2>/dev/null | ./xmlgrep3.py item/title
    item/description
[item/title] Interview : Rencontre avec Taguan
[item/description] Une lead dev' au pays de la bière et des frites
    !
[item/title] Programmez en langage d'assemblage sous Linux !
[item/description] Prise en main de nasm et découverte du langage
    d'assemblage x86 / x64
[item/title] Zeste de Savoir passe en version 1.7
[item/description] Tout ce qu'il faut savoir sur cette nouvelle
    version (et la 1.6)
[item/title] Assemblée générale de l'association Zeste de Savoir
[item/description] Parce que ça mérite bien un compte rendu
[item/title] La géographie, une approche pour comprendre le monde
    qui nous entoure
[item/description] Présentation d'une discipline trop méconnue
```

3.4. Pour aller plus loin

Je disais plus haut que cette machine à états avait un comportement contre-intuitif. En fait, les chemins qu'elle comprend sont différents des *xpaths* tels qu'on les connaît. La première différence est qu'en l'état, le séparateur `/` de nos chemins correspondrait plus ou moins à un séparateur récursif `//` dans la syntaxe xpath.

La piste d'amélioration la plus immédiate pour `xmlgrep` est donc d'essayer d'implémenter une syntaxe des chemins compatible xpath. La plupart des modifications à apporter à ce code concernent surtout la méthode `add_path`, puisqu'il suffit, en réalité, de rajouter des transitions vers de nouveaux états de la machine actuelle.

D'autres composantes de la syntaxe xpath pourraient s'avérer pratiques, comme les tests sur la valeur des attributs d'un noeud :

```
path: /rss/item[type=article]/title

<rss>
    <item type="article">
        <title>Ce titre sera extrait</title>
    </item>
    <item>
        <title>Celui-ci doit être ignoré</title>
    </item>
</rss>
```

Mais là encore, je me permets de réutiliser cette phrase que j'adore écrire car elle est pompeuse au possible : *l'implémentation de ces fonctionnalités est laissée au lecteur en guise d'exercice.*

3.5. Et niveau perfs, alors ?

Pour conclure, voyons un peu comment les coroutines s'en sortent face à d'autres parseurs XML. Commençons par le résultat le plus évident : l'économie de mémoire.

J'ai fait manger un fichier XML de 45 Mio à `xmlgrep` ainsi qu'à un parseur plus classique qui produit un arbre avant de faire la recherche par `xpath`, que j'appellerai `xmltree` :

```
1 import sys
2 from xml.etree.ElementTree import parse
3
4 tree = parse(sys.stdin)
5 for elt in tree.iterfind('.//item//title'):
6     print(elt.text)
```

`xmlgrep` s'en est sorti avec une utilisation globale de 10,4 Mio de mémoire contre… 221,3 Mio pour `xmltree`. De ce point de vue-là, les coroutines remportent le match haut la main !

Maintenant, côté performances.

J'ai d'abord comparé `xmlgrep`, en commentant les `print()` pour lisser l'impact des IO sur la mesure, avec la version Python 2 de `ElementTree`, puis sa version un peu plus optimisée en Python 3, sur la même donnée d'entrée gargantuesque :

— `xmlgrep` : 5,195 s
— `xmltree (Python 2)` : 11,331 s
— `xmltree (Python 3)` : 2,591 s

Jusqu'ici, on s'aperçoit que `xmlgrep` est environ deux fois plus rapide que le parseur pur Python de Python 2, mais deux fois plus lent que celui de Python 3.

Néanmoins, après un rapide profilage, il s'est avéré que le programme passait *la moitié de son temps* dans le parseur à produire des événements. J'ai donc remplacé la fonction `iterparse` de la bibliothèque standard par celle, beaucoup plus performante, de `lxml` , qui permet notamment de pré-filtrer les balises dont on souhaite écouter les événements :

— `xmlgrep (lxml)` : 1,971 s
— `xmltree (lxml)` : 1,789 s

Soit une différence de l'ordre d'à peine 10% pour une recherche simultanée de deux clés, par rapport à la solution la plus rapide possible en Python (qui repose sur une bibliothèque en C).

Cela dit, si l'on décommente les `print()`, qui sont objectivement les opérations les plus coûteuses dans les deux cas puisque la sortie prend 79200 lignes, on s'aperçoit que cette différence est fluctuante, et qu'à mesure que la donnée grossit, la balance a même tendance à pencher en faveur de `xmlgrep` :

— `xmlgrep (lxml, with IO)` : 12,430 s
— `xmltree (lxml, with IO)` : 13,221 s

Sans aller jusqu'à fanfaronner, nous pouvons en conclure que `xmlgrep`, dont l'empreinte mémoire reste constante en toutes circonstances, est globalement *aussi rapide* qu'un programme plus classique utilisant `lxml`, dont la mémoire explose avec la taille de la donnée d'entrée.

4. Conclusion

Nous venons de découvrir un puissant sortilège de magie noire en Python. Comme nous avons pu l'observer, les coroutines se prêtent bien à des implémentations à la fois simples, élégantes et performantes de programmes incrémentaux. Mais nous sommes encore **très** loin d'en avoir fait le tour !

En particulier, nous n'avons pas encore abordé l'implémentation des boucles événementielles du style de la bibliothèque `asyncio`. Patience : ce sera pour la prochaine fois !