SORTIE DE PYTHON 3.5

artragis, nohar, Kje, Vayel, Emeric

30 octobre 2015

Zeste de savoir

Table des matières

1 Introduction

Une nouvelle version du langage Python (et par conséquent de son implémentation principale CPython) sort aujourd'hui. Estampillée 3.5, cette version fait logiquement suite à la version 3.4 parue il y a un an et demi[1]. Tandis que cette dernière apportait principalement des ajouts dans la bibliothèque standard (asyncio, enum, ensurepip, pathlib, etc.), les nouveautés les plus visibles de la version 3.5 concernent des changements syntaxiques avec deux nouveaux mot-clés, un nouvel opérateur binaire, la généralisation de l'*unpacking* et la standardisation des annotations de fonctions.

Cette version 3.5 sera certainement perçue par les Pythonistes comme celle qui aura introduit le plus de changements au langage depuis Python 3.0. En effet, nous allons découvrir ensemble qu'avec cette nouvelle version, Python achève d'inclure dans le langage le support d'un paradigme de programmation moderne, revenu au goût du jour avec l'éclosion de Node.js : la programmation asynchrone.

1. Le 16 Mars 2014 pour être précis.

2 TL;DR - Résumé des principales nouveautés

Les plus pressés peuvent profiter du court résumé des principales nouveautés suivant. Chacune sera reprise dans la suite de l'article, en la détaillant et explicitant les concepts sous-jacents.

— PEP 448 : les opérations d'*unpacking* sont généralisées et peuvent maintenant être combinées et utilisées plusieurs fois dans un appel de fonction.

```
>>> def spam(a, b, c, d, e, g, h, i, j):
...     return a + b + c + d + e + g + h + i + j
...
>>> l1 = (1, *[2])
>>> l1
(1, 2)
>>> d1 = {"j" : 9, **{"i" : 8}}
>>> d1
{"j" : 9, "i" : 8}
>>> spam(*l1, 3, *(4, 5), g=6, **d1, **{'h' :7})
45    # 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9
```

— PEP 465 : l'opérateur binaire `@` est introduit pour gérer la multiplication matricielle et permet d'améliorer la lisibilité d'expressions mathématiques. Par exemple, l'équation $A \times B^T - C^{-1}$ correspondrait au code suivant avec `numpy`.

```
A @ B.T - inv(C)
```

— PEP 492 : les coroutines deviennent une construction spécifique du langage, avec l'introduction des mots-clés `async` et `await`. L'objectif étant de compléter le support de la programmation asynchrone dans Python, ils permettent d'écrire des coroutines utilisables avec `asyncio` de façon similaire à des fonctions Python synchrones classiques. Par exemple :

```
async def fetch_page(url, filename):
    # L'appel à aiohttp.request est asynchrone
    response = await aiohttp.request('GET', url)
    # Ici, on a récupéré le résultat de la requête
    assert response.status == 200
    async with aiofiles.open(filename, mode='wb') as fd:
        async for chunk in response.content.read_chunk(chunk_size):
            await fd.write(chunk)
```

— PEP 484 : les annotations apposables sur les paramètres et la valeur de retour des fonctions et méthodes sont maintenant standardisées et ne devraient servir qu'à préciser le type de

ces éléments. Les annotations ne sont toujours pas utilisées par l'interpréteur et cette PEP n'est constituée que de conventions.

```
def bonjour(nom: str) -> str :
    return 'Zestueusement ' + nom
```

3 Principales nouveautés

3.1 Unpacking généralisé – PEP 448

L'*unpacking* est une opération permetant de séparer un itérable en plusieurs variables :

```
>>> l = (1, 2, 3, 4)
>>> a, b, *c = l
>>> a
1
>>> b
2
>>> c
(3, 4)
>>> def f(a, b):
...     return a + b, a - b    # f retourne un tuple
...
>>> somme, diff = f(5, 3)
>>> somme
8    # 5 + 3
>>> diff
2    # 5 - 3
```

Il est alors possible de passer un itérable comme argument de fonction comme si son contenu était passé élément par élément grâce à l'opérateur * ou de passer un dictionnaire comme arguments nommés avec l'opérateur ** :

```
>>> def spam(a, b):
...     return a + b
...
>>> l = (1, 2)
>>> spam(*l)
3    # 1 + 2
>>> d = {"b" : 2, "a" : 3}
>>> spam(**d)
5    # 3 + 2
```

Cette fonctionnalité restait jusqu'à maintenant limitée et les conditions d'utilisation très strictes. Deux de ces contraintes ont été levées...

3.1.1 Support de l'unpacking dans les déclarations d'itérables

Lorsque vous souhaitez définir un `tuple`, `list`, `set` ou `dict` littéral, il est maintenant possible d'utiliser l'*unpacking*.

```
## Python 3.4
>>> tuple(range(4)) + (4,)
(0, 1, 2, 3, 4)
## Python 3.5
>>> (*range(4), 4)
(0, 1, 2, 3, 4)

## Python 3.4
>>> list(range(4)) + [4]
[0, 1, 2, 3, 4]
## Python 3.5
>>> [*range(4), 4]
[0, 1, 2, 3, 4]

## Python 3.4
>>> set(range(4)) + {4}
set(0, 1, 2, 3, 4)
## Python 3.5
>>> {*range(4), 4}
set(0, 1, 2, 3, 4)

## Python 3.4
>>> d = {'x' : 1}
>>> d.update({'y' : 2})
>>> d
{'x' : 1, 'y' : 2}
## Python 3.5
>>> {'x' : 1, **{'y' : 2}}
{'x' : 1, 'y' : 2}

## Python 3.4
>>> combinaison = long_dict.copy()
>>> combination.update({'b' : 2})
## Python 3.5
>>> combinaison = {**long_dict, 'b' : 2}
```

Notamment, il devient maintenant facile de sommer des itérables pour en former un autre.

```
>>> l1 = (1, 2)
>>> l2 = [3, 4]
>>> l3 = range(5, 7)

## Python 3.5
>>> combinaison = [*l1, *l2, *l3, 7]
>>> combinaison
[1, 2, 3, 4, 5, 6, 7]

## Python 3.4
>>> combinaison = list(l1) + list(l2) + list(l3) + [7]
```

```
>>> combinaison
[1, 2, 3, 4, 5, 6, 7]
```

Cette dernière généralisation permet ainsi d'apporter une symétrie par rapport aux possibilités précédentes.

```
## Possible en Python 3.4 et 3.5
>>> elems = [1, 2, 3, 4]
>>> fst, *other, lst = elems
>>> fst
1
>>> other
[2, 3]
>>> lst
4

## Possible uniquement en Python 3.5
>>> fst = 1
>>> other = [2, 3]
>>> lst = 4
>>> elems = fst, *other, lst
>>> elems
(1, 2, 3, 4)
```

3.1.2 Support de plusieurs unpacking dans les appels de fonctions

Cette généralisation se répercute sur les appels de fonctions ou méthodes : jusqu'à Python 3.4, un seul itérable pouvait être utilisé lors de l'appel à une fonction. Cette restriction est maintenant levée.

```
>>> def spam(a, b, c, d, e):
...     return a + b + c + d + e
...
>>> l1 = (2, 1)
>>> l2 = (4, 5)
>>> spam(*l1, 3, *l2)         # Légal en Python 3.5, impossible en Python 3.4
15                            # 2 + 1 + 3 + 4 + 5
>>> d1 = {"b" : 2}
>>> d2 = {"d" : 1, "a" : 5, "e" : 4}
>>> spam(**d1, c=3, **d2)     # Légal en Python 3.5, impossible en Python 3.4
15                            # 5 + 2 + 3 + 1 + 4
>>> spam(*(1, 2), **{"c" : 3, "e" : 4, "d" : 5})
15                            # 1 + 2 + 3 + 5 + 4
```

Notez que si un nom de paramètre est présent dans plusieurs dictionnaires, c'est la valeur du dernier qui sera prise en compte.

D'autres généralisations sont mentionnées dans la PEP mais n'ont pas été implémentées à cause d'un manque de popularité parmi les developpeurs de Python. Il n'est cependant pas impossible que d'autres fonctionnalités soient introduites dans les prochaines versions.

3.2 Opérateur de multiplication matricielle – PEP 465

L'introduction d'un opérateur binaire n'est pas courant dans Python. Aucun dans la série 3.x, le dernier ajout semble être l'opérateur // dans Python 2.2[1]. Regardons donc pour quelles raisons celui-ci a été introduit.

3.2.1 Signification

Ce nouvel opérateur est dédié à la multiplication matricielle. En effet, comme tous ceux ayant fait un peu de mathématiques algébriques le savent sûrement, on définit généralement la multiplication matricielle par l'opération suivante (pour des matrices $2 * 2$ ici) :

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \cdot \begin{pmatrix} e & f \\ g & h \end{pmatrix} = \begin{pmatrix} a*e+b*g & a*f+b*h \\ c*e+d*g & c*f+d*h \end{pmatrix}$$

Or il est souvent aussi nécessaire d'effectuer des multiplications terme à terme :

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} * \begin{pmatrix} e & f \\ g & h \end{pmatrix} = \begin{pmatrix} a*e & b*f \\ c*e & d*h \end{pmatrix}$$

Tandis que certains langages spécialisés possèdent des opérateurs dédiés pour chacune de ces opérations[2] il n'y a en Python rien de similaire. Avec la bibliothèque numpy, la plus populaire pour le calcul numérique dans l'éco-système Python, il est possible d'utiliser l'opérateur natif * pour effectuer une multiplication terme à terme, surcharge d'opérateur se rencontrant dans la plupart des bibliothèques faisant intervenir les matrices. Mais il n'existe ainsi plus de moyen simple pour effectuer une multiplication matricielle. Avec `numpy`, il est pour le moment nécessaire d'utiliser la méthode `dot` et d'écrire des lignes de la forme :

```
S = (A.dot(B) - C).T.dot(inv(A.dot(D).dot(A.T))).dot(A.dot(B) - C)
```

Celle-ci traduit cette formule :

$$(A \times B - C)^T \times (A \times D \times A^T)^{-1} \times (A \times B - C)$$

Cela n'aide pas à la lecture… Le nouvel opérateur @ est introduit et dédié à la multiplication matricielle. Il permettra d'obtenir des expressions équivalentes de la forme :

```
S = (A @ B - C).T @ inv(A @ D @ A.T) @ (A @ B - C)
```

Un peu mieux, non ?

1. Ces ajouts sont tellement peu courants qu'il est difficile de trouver des traces de ces modifications. L'opérateur // semble être le seul ajout de toute la série 2.
2. Par exemple, avec Matlab et Julia, * / .* servent respectivement pour la multiplication matricielle et terme à terme.

3.2.2 Impact sur vos codes

Cette introduction devrait être anodine pour beaucoup d'utilisateurs. En effet, aucun objet de la bibliothèque standard ne va l'utiliser[3]. Cet opérateur binaire servira principalement à des bibliothèques annexes, à commencer par *numpy*.

Si vous souhaitez supporter cet opérateur, trois méthodes spéciales peuvent être implémentées : `__matmul__` et `__rmatmul__` pour la forme `a @ b` et `__imatmul__` pour la forme `a @= b`, de façon similaire aux autres opérateurs binaires.

À noter qu'il est déconseillé d'utiliser cet opérateur pour autre chose que les multiplications matricielles.

3.2.3 Motivation

L'introduction de cet opérateur est un modèle du genre : il peut sembler très spécifique mais est pleinement justifié. La lecture de la PEP, développée, est très instructive.

Pour la faire adopter, les principales bibliothèques scientifiques en Python ont préparé cette PEP ensemble pour arriver à une solution convenant à la grande partie de la communauté scientifique, assurant dès lors l'adoption rapide de cet opérateur. La PEP précise ainsi l'intérêt et les problèmes engendrés par les autres solutions utilisées jusque-là.

Enfin cette PEP a été fortement appuyée par la grande popularité de Python dans le monde scientifique. On apprend aussi que `numpy` est le module n'appartenant pas à la bibliothèque standard le plus utilisé parmi tous les codes Python présents sur Github et ce sans compter d'autres bibliothèques comme `pylab` ou `scipy` qui vont aussi profiter de cette modification et comptent parmi les bibliothèques les plus communes.

3.3 Support des coroutines – PEP 492

3.3.0.1 Introduction à la programmation asynchrone

Pour comprendre ce qu'est la programmation asynchrone, penchons-nous sur l'exemple de code suivant, chargé de télécharger le contenu de la page d'accueil de *Zeste de Savoir* et de l'enregistrer sur le disque par morceaux de 512 octets en utilisant la bibliothèque requests.

```
import requests

def fetch_page(url, filename, chunk_size=512):

    # Nous effectuons une demande de connexion au serveur HTTP et attendons sa réponse
    response = requests.get(url, stream=True)

    # Nous vérifions qu'il n'y a pas eu d'erreur lors de la connexion
    assert response.status_code == 200

    # Nous ouvrons le fichier d'écriture
```

3. Ce qui est rare mais existe déjà. En particulier l'objet `Elipsis` créé avec `...`.

```
    with open(filename, mode='wb') as fd:

        # Nous récupérons le fichier de la page d'accueil par morceaux : à chaque
        # tour de boucle, nous demandons un paquet réseau au serveur et attendons
        # qu'il nous le donne
        for chunk in response.iter_content(chunk_size=chunk_size):

            # Nous écrivons dans le fichier de sortie, sur le disque
            fd.write(chunk)

if __name__ == "__main__" :
    fetch_page('http://zestedesavoir.com/', 'out.html')
```

Lors d'un appel à ce genre de fonction très simple votre processeur passe son temps… à ne rien faire ! En effet, quand il n'attend pas le serveur Web, il patiente le temps que le disque effectue les opérations d'écriture demandées. Or tout cela est très lent à l'échelle d'un processeur, et tous ces appels à des services extérieurs sont très courants en Web : connexion et requêtes à une base de données, appels à une API externe, etc. La perte de temps est de ce fait considérable.

La programmation dite **asynchrone** cherche à résoudre ce problème en permettant au développeur d'indiquer les points où la fonction attend un service extérieur (base de données, serveur Web, disque dur, etc.), appelé *entrées/sorties* (en anglais : *io*). Le programme principal peut ainsi faire autre chose pendant ce temps, comme traiter la requête d'un autre client. Pour cela une boucle évenementielle est utilisée. Le coeur de l'application est alors une fonction qu'on pourrait résumer par les opérations suivantes :

1. Prendre une tâche disponible
2. Exécuter la tâche jusqu'à ce qu'elle soit terminée ou qu'elle doive attendre des entrées/sorties
3. Dans le second cas, la mettre dans une liste de tâches en attente
4. Mettre dans la liste des tâches disponibles les tâches en attente ayant reçu leurs données
5. Retourner en 1

Prenons un exemple, avec un serveur Web :

— Un client `A` demande une page, appelant ainsi une fonction `f`
— Un client `B` demande une page, appelant une autre fonction `g`, appel mis en attente vu que le processeur est occupé avec le client `A`
— L'appel à `f` atteint une écriture dans un fichier, et se met en pause le temps que le disque fasse son travail
— `g` est démarrée
— L'écriture sur le disque de `f` est terminée
— `g` atteint une demande de connexion à la base de données et se met en pause le temps que la base réagisse
— L'appel à `f` est poursuivi et terminé
— Un client `C` demande une page
— etc.

En découpant une fonction par morceaux, la boucle peut exécuter du code pendant que les longues opérations d'entrées/sorties d'un autre morceau de code se déroulent à l'extérieur…

[[i]] | Nous nous comportons naturellement de cette manière dans la vie. Par exemple, | lorsque vous effectuez un rapport que vous devez faire relire à votre chef : après | lui avoir fait parvenir une première version, vous allez devoir attendre qu'il | l'ait étudié avant de le corriger ; plutôt que de patienter bêtement devant son | bureau, vous vaquez à vos occupations, et serez informé lorsque le rapport pourra | être repris.

3.3.0.2 La programmation asynchrone en Python

La programmation asynchrone a été remise récemment au goût du jour par Node.js. Mais Python n'est pas en reste, avec de multiples bibliothèques pour gérer ce paradigme : Twisted existe depuis 13 ans (2002), Tornado a été libérée par FriendFeed il y a 6 ans (2009), à l'époque où le développement de gevent commençait.

Au moment de la version 3.4 de Python, Guido van Rossum, créateur et BDFL du langage, a décidé de standardiser cette approche en synthétisant les idées des bibliothèques populaires précédemment citées et d'intégrer une implémentation typique dans la bibliothèque standard : c'est le module asyncio. Ce dernier n'a pas pour but de remplacer les bibliothèques mentionnées, mais de proposer une implémentation standardisée d'une boucle événementielle et de mettre à disposition quelques fonctionnalités bas niveau (*Tornado* peut ainsi être utilisé avec asyncio).

[BDFL] : Benevolent Dictator For Life* (dictateur bienveillant à vie)

Tandis que *Node.js*, par exemple, emploie ce que l'on appelle des fonctions de rappel (*callbacks*) pour ordonnancer les différentes étapes d'un algorithme, *asyncio* utilise des coroutines, lesquelles permettent d'écrire des fonctions asynchrones avec un style procédural. Les coroutines ressemblent beaucoup aux fonctions, à la différence près que leur exécution peut être suspendue et reprise à plusieurs endroits dans la fonction. Python dispose déjà de constructions de ce genre : les générateurs [4]. C'est ainsi avec les générateurs que *asyncio* a été initialement développé.

Pour illustrer cela, reprenons l'exemple décrit plus haut, avec cette fois Python 3.4, *asyncio*, la bibliothèque aiohttp [5] pour faire des requêtes HTTP et la bibliothèque aiofiles pour écrire dans des fichiers locaux, le tout de façon asynchrone :

```
import asyncio
import aiohttp
import aiofiles

@asyncio.coroutine           # On déclare cette fonction comme étant une coroutine
def fetch_page(url, filename, chunk_size=512):
    # À la ligne suivante, la fonction est interrompue tant que la requête n'est pas r
    # Le programme peut donc vaquer à d'autres tâches
    response = yield from aiohttp.request('GET', url)

    # Le serveur HTTP nous a répondu, on reprend l'appel de la fonction à ce
    # niveau et vérifie que la connexion est correctement établie
    assert response.status == 200

    # De même, on suspend ici la fonction le temps que le fichier soit créé
```

4. Les générateurs sont en théorie des formes particulières de coroutines, mais leur implémentation en Python leur confère pleinement le statut de coroutine.
5. Bibliothèque qui propose des fonctions d'entrées/sorties sur le protocole HTTP.

```
    fd = yield from aiofiles.open(filename, mode='wb')
    try :
        while True :
            # On demande un paquet au serveur Web et interrompt la coroutine le
            # temps qu'il nous réponde
            chunk = yield from response.content.read(chunk_size)

            if not chunk:
                break

            # On suspend la coroutine le temps que le disque dur écrive dans le fichie
            yield from fd.write(chunk)

    finally :
        # On ferme le fichier de manière asynchrone
        yield from fd.close()

if __name__ == "__main__" :
    # On démarre une tâche dans la boucle évènementielle.
    # Vu qu'il n'y en a qu'une seule, procéder de manière asynchrone n'est pas très ut
    asyncio.get_event_loop().run_until_complete(fetch_page('http://zestedesavoir.com/'
```

Cet exemple nous montre comment utiliser les générateurs et *asyncio* pour obtenir un code asynchrone lisible. Toutefois :

— Il y a détournement du rôle d'origine de l'instruction `yield from`. Sans le décorateur la fonction pourrait être facilement confondue avec un générateur classique, sans rapport avec de l'asynchrone.
— Tandis que l'exemple d'origine utilisait `with` pour assurer la fermeture du fichier même en cas d'exception, nous sommes obligés ici de reproduire manuellement le comportement de cette instruction. En effet `with` n'est pas prévue pour appeler des coroutines, donc on ne peut effectuer de manière asynchrone les opérations d'entrée (ligne 16) ni de sortie (ligne 31).
— De la même façon, l'instruction `for` ne permet pas de lire un itérable de manière asynchrone, comme fait ligne 21. Il nous faut donc passer peu élégamment par une boucle `while`.

3.3.1 Les nouveaux mot-clés

Python 3.5 introduit deux nouveaux mot-clés pour résoudre les problèmes précédemment cités : `async` et `await`. De l'extérieur, `async` vient remplacer le décorateur `asyncio.coroutine` et `await` l'expression `yield from`.

Mais les modifications sont plus importantes que ces simples synonymes. Tout d'abord, une coroutine déclarée avec `async` n'est **pas** un générateur. Même si en interne les deux types de fonctions partagent une grande partie de leur implémentation, il s'agit de constructions du langage différentes et il est possible que les différences se creusent dans les prochaines versions de Python. Pour marquer cette distinction et le fait que des générateurs sont une forme restreinte de coroutines, il est possible dans Python 3.5 d'utiliser des générateurs partout où une coroutine est attendue, mais pas l'inverse. Le module *asyncio* continue ainsi à supporter les deux formes.

L'ajout de ces mot-clés a aussi été l'occasion d'introduire la possibilité d'itérer de manière asynchrone sur des objets. Ce support, assuré en interne par les nouvelles méthodes `__aiter__` et `__anext__` pourra être utilisé par des bibliothèques comme *aiohttp* pour simplifier les itérations grâce à la nouvelle instruction `async for`.

De la même façon, des *context managers* asynchrones font leur apparition via l'instruction `async with`, en utilisant les méthodes `__aenter__` et `__aexit__`.

Il est donc possible, et conseillé, en Python 3.5 de ré-écrire le code précédent de la manière suivante.

```
import asyncio
import aiohttp
import aiofiles

## On déclare cette fonction comme étant une coroutine
async def fetch_page(url, filename, chunk_size=512):
    # À la ligne suivante, la fonction est interrompue tant que la connexion n'est pas
    response = await aiohttp.request('GET', url)
    assert response.status == 200

    # Ouverture, puis fermeture, du fichier de sortie de manière asynchrone
    async with aiofiles.open(filename, mode='wb') as fd:

        # On lit le contenu au fur et à mesure que les paquets arrivent et on interrom
        async for chunk in response.content.read_chunk(chunk_size):

            await fd.write(chunk)

if __name__ == "__main__" :
    asyncio.get_event_loop().run_until_complete(fetch_page('http://zestedesavoir.com/'
```

L'exemple ci-dessus montre clairement l'objectif de l'ajout de `async for` et `async with` : si les `async` et `await` sont ignorés, la coroutine est fortement similaire à une implémentation utilisant des fonctions classiques présentées en début de section.

[[a]] | Le dernier exemple de codes est un aperçu de ce que pourrait être la | programmation asynchrone avec Python 3.5. À l'heure où ces lignes sont écrites, | *aiohttp* et *aiohttp* ne supportent pas encore les nouvelles instructions `async for` | et `async with`. De la même manière, pour l'instant, seul *asyncio* gère les mot-clés | `async` et `await` au niveau de sa boucle événementielle.

Enfin, notez que les expressions `await`, au-delà de leur précédence beaucoup plus faible, sont moins restreintes que les `yield from` et peuvent être placées partout où une expression est attendue. Ainsi les codes suivants sont valides.

```
if await foo:
    pass

## /!\ Différent de `async with`
# Ici, c'est `bar` qui est appelé de manière asynchrone et non `bar.__enter__`
with await bar:
     pass
```

```
while await spam:
     pass

await spam + await egg
```

3.3.2 Impact sur vos codes

Concernant vos codes existants, cette version introduit logiquement la dépréciation de l'utilisation de `async` et `await` comme noms de variable. Pour le moment, un simple avertissement sera émis, lequel sera transformé en erreur dans Python 3.7. Ces modifications restent donc parfaitement rétro-compatibles avec Python 3.4.

Si vous utilisez *asyncio*, rien ne change pour vous et vous pouvez continuer à employer les générateurs si vous souhaitez conserver le support de Python 3.4. L'ajout des méthodes de la forme `__a****__` n'est donc pas obligatoire, mais peut vous permettre cependant de supporter les nouveautés de Python 3.5.

3.4 Annotations de types – PEP 484

Les annotations de fonctions existent depuis Python 3 et permettent d'attacher des objets Python aux arguments et à la valeur de retour d'une fonction ou d'une méthode :

```
def ma_fonction(param: str, param2: 42) -> ["Mon", "annotation"]:
    pass
```

Depuis le début, il est possible d'utiliser n'importe quel objet valide comme annotation. Ce qui peut surprendre, c'est que l'interpréteur n'en fait pas d'utilisation particulière : il se contente de les stocker dans l'attribut `__annotations__` de la fonction correspondante :

```
>>> ma_fonction.__annotations__
{'param2' : 42, 'return' : ['Mon', 'annotation'], 'param' : <class 'str'>}
```

La PEP 484 introduite avec Python 3.5 fixe des conventions pour ces annotations : elles deviennent réservées à « l'allusion de types » (*Type Hinting*), qui consiste à indiquer le type des arguments et de la valeur de retour des fonctions :

```
## Cette fonction prend en argument une chaîne de caractères et en retourne une autre.
def bonjour(nom: str) -> str :
    return 'Zestueusement ' + nom
```

Toutefois, il ne s'agit encore que de conventions : l'interpréteur Python ne fera rien d'autre que de les stocker dans l'attribut `__annotations__`. Il ne sera même pas gêné par une annotation d'une autre forme qu'un type.

Pour des indications plus complètes sur les types, un module `typing` est introduit. Il permet de définir des types génériques, des tableaux, etc. Il serait trop long de détailler ici toutes les possibilités de ce module, donc nous nous contentons de l'exemple suivant.

```
## On importe depuis le module typing :
# - TypeVar pour définir un ensemble de types possibles
## - Iterable pour décrire un élément itérable
# - Tuple pour décrire un tuple
from typing import TypeVar, Iterable, Tuple

## Nous définissons ici un type générique pouvant être un des types numériques cités.
T = TypeVar('T', int, float, complex)
## Vecteur représente un itérateur (list, tuple, etc.) contenant des tuples
# comportant chacun deux éléments de type T.
Vecteur = Iterable[Tuple[T, T]]

## Pour déclarer un itérable de tuples, sans spécifier le contenu de ces derniers,
# nous pouvons utiliser le type natif :
## Vecteur2 = Iterable[tuple]
# Les éléments présents dans le module typing sont là pour permettre une
## description plus complète des types de base.

## Nous définissons une fonction prenant un vecteur en argument et renvoyant un nombre
def inproduct(v: Vecteur) -> T:
    return sum(x*y for x, y in v)

vec = [(1, 2), (3, 4), (5, 6)]
res = inproduct(vec)
## res == 1 * 2 + 3 * 4 + 5 * 6 == 44
```

Néanmoins, les annotations peuvent surcharger les déclarations et les rendre peu lisibles. Cette PEP a donc introduit une convention supplémentaire : les fichiers `Stub`. Il a été convenu que de tels fichiers, facultatifs, contiendraient les annotations de types, permettant ainsi de profiter de ces informations sans polluer le code source. Par exemple, le module `datetime` de la bibliothèque standard comporterait un fichier `Stub` de la forme suivante.

```
class date(object):
    def __init__(self, year: int, month: int, day: int): ...
    @classmethod
    def fromtimestamp(cls, timestamp: int or float) -> date: ...
    @classmethod
    def fromordinal(cls, ordinal: int) -> date: ...
    @classmethod
    def today(self) -> date: ...
    def ctime(self) -> str : ...
    def weekday(self) -> int : ...
```

Ces fichiers permettent aussi d'annoter les fonctions de bibliothèques écrites en C ou de fournir des annotations de types aux modules utilisant déjà les annotations pour autre chose que le *type hinting*.

Tout comme rien ne vous oblige à utiliser les annotations pour le typage, rien ne vous force à vous servir du module `typing` ou des fichiers `Stub` : l'interpréteur n'utilisera ni ne vérifiera toujours pas ces annotations ; le contenu des fichiers `Stub` ne sera même pas intégré à l'attribut __an-

`notations__`. Ne vous attendez donc pas à une augmentation de performances en utilisant les annotations de types : l'objectif de cette PEP est de normaliser ces informations pour permettre à des outils externes de les utiliser. Les premiers bénéficiaires seront donc les EDI, les générateurs de documentation (comme Sphinx) ou encore les analyseurs de code statiques comme mypy, lequel a fortement inspiré cette PEP et sera probablement le premier outil à proposer un support complet de cette fonctionnalité.

Les annotations de types ne sont donc qu'un ensemble de conventions, comme il en existe déjà plusieurs dans le monde Python (PEP 333, PEP 8, PEP 257, etc.). Cet ajout n'a donc aucun impact direct sur vos codes mais permettra aux outils externes de vous fournir du support supplémentaire.

4 De plus petits changements

— PEP 441 : ajout d'un module `zipapp`, pour améliorer le support et la création d'applications packagées sous forme d'archive `zip`, introduits dans Python 2.6.
— PEP 461 : retour de l'opérateur de formatage `%` pour les types de données `byte` et `bytearray`, de la même façon qu'il était utilisable pour les types chaînes (`str`) dans Python 2.7.
— PEP 488 : la suppression des fichiers `pyo`.
— PEP 489 : un changement de la procédure d'import des modules externes.
— PEP 471 : une nouvelle fonction `os.scandir()`, pour itérer plus efficacement sur le contenu d'un dossier sur le disque qu'en utilisant `os.walk()`. Cette dernière fonction l'utilise maintenant en interne et est de trois à cinq fois plus rapide sur les systèmes POSIX et de sept à vingt fois plus rapide sur Windows.
— PEP 475 : l'interpréteur va maintenant automatiquement retenter le dernier appel système lorsqu'un signal `EINTR` est reçu, évitant aux codes utilisateurs de s'en occuper.
— PEP 479 : le comportement des générateurs changent lorsqu'une exception `StopIteration` est déclenchée à l'intérieur. Avec cette PEP, cette exception est transformée en `RuntimeError` plutôt que de quitter silencieusement le générateur. Cette modification cherche à faciliter le déboguage et clarifie la façon de quitter un générateur : utiliser `return` plutôt que de générer une exception. Cette PEP n'étant pas rétro-compatible, vous devez manuellement l'activer avec `from __future__ import generator_stop` pour en profiter.
— PEP 485 : une fonction `isclose` est ajoutée pour tester la « proximité » de deux nombres flottants.
— PEP 486 : le support de `virtualenv` dans l'installateur de Python sous Windows est amélioré et simplifié.
— Ticket 9951 : pour les types `byte` et `bytearray`, une nouvelle méthode `hex()` permet maintenant de récupérer une chaîne représentant le contenu sous forme hexadécimale.
— Ticket 16991 : l'implémentation en C, plutôt qu'en Python, des `OrderedDict` (dictionnaires ordonnés) apporte des gains de quatre à cent sur les performances.

De plus, comme d'habitude, tout un tas de fonctions, de petites modifications et de corrections de bugs ont été apportées à la bibliothèque standard, qu'il serait trop long de citer entièrement.

Notons aussi que le support de Windows XP est supprimé. Cela ne veut pas dire que Python 3.5 ne fonctionnera pas dessus, mais rien ne sera fait par les développeurs pour cela.

Enfin, le développement de CPython 3.5 (à partir de la première *release* candidate) s'est effectué sur BitBucket plutôt que sur le traditionnel dépôt auto-hébergé de la fondation, afin de bénéficier des fonctionnalités proposées par le site ; le gestionnaire de versions est toujours Mercurial. Il n'est pas encore connu si les prochaines versions suivront ce changement et si le développement de CPython se déplacera sur BitBucket : il s'agit d'une expérimentation que les développeurs vont évaluer. Pour le moment, les branches pour les versions 3.5.1 et 3.6 restent sur https://hg.python.org/cpython au moins jusqu'à la sortie de Python 3.5.

5 Ce que l'on peut attendre pour la version 3.6

Il est impossible de savoir précisément ce qui sera disponible dans la prochaine version du langage et de l'interpréteur. En effet aucune entreprise ou groupe ne décide par avance des fonctionnalités. Les changements inclus dépendent des propositions faites, majoritairement, sur deux *mailling list* :

— *python-dev* quand cela concerne des changements internes à l'intepréteur CPython.
— *python-ideas* quand cela concerne des idées générales à propos du langage.

S'ensuit une série de débats qui, si les idées sont acceptées par une majorité de developpeurs principaux, conduit à la rédaction d'une PEP, qui sera elle-même acceptée ou refusée (par Guido, le BDFL, ou par un autre développeur si Guido est l'auteur de la PEP). En l'absence de feuilles de route définies à l'avance, les PEP peuvent arriver tard dans le cycle de développement. Ainsi, la PEP 492 sur `async` et `await` n'a été créé qu'un mois avant la sortie de la première bêta de Python 3.5.

[[a]] | Malgré ces incertitudes, on peut deviner quelques modifications probables. Attention, cette section reste très spéculative…

5.1 Continuité des changements introduits dans Python 3.5

Trois thématiques de modifications amorcées dans Python 3.4 et surtout Python 3.5 pourraient être de nouveau sources d'ajouts importants dans Python 3.6 :

— La programmation asynchrone avec *asyncio* et les coroutines
— Les indications de types
— La généralisation de l'*unpacking*

Les deux premiers éléments sont officiellement en attente de retours de la part des utilisateurs de Python. Les développeurs de l'interpréteur attendent de connaître les problèmes et limitations rencontrées en utilisant ces fonctionnalités pour les peaufiner dans les prochaines versions. Python 3.6 devrait donc logiquement voir des améliorations dans ces deux sections en profitant des retours. Déjà quelques remarques ont été faites, comme la complexité de mêler des fonctions synchrones et asynchrones.

La généralisation de l'*unpacking* pourrait elle aussi continuer. La PEP 448 proposait initialement d'autres généralisations qui n'ont pas été retenues pour Python 3.5 par manque de temps. Elles seront donc propablement rapidement rediscutées. De plus, ces ajouts dans Python 3.5 ont donné des idées à d'autres développeurs et quelques nouvelles modifications ont été déjà proposées.

5.2 Conservation de l'ordre des arguments fournis à **kwargs lors de l'appel aux fonctions

Cette PEP a déjà été acceptée mais n'a pas pu être implémentée dans la version 3.5. Il est donc possible qu'elle soit finalement présente dans la prochaine version. L'idée est que les fonctions puissent connaître l'ordre dans lequel les arguments nommés ont été passés. Prenons un exemple :

```
def spam(**kwargs):
    for k, v in kwargs.items():
        print(k, v)

spam(a=1, b=2)
```

Avec Python 3.5, comme toutes versions de CPython et presque toutes les autres implémentations (sauf PyPy), nous ne pouvons pas savoir si le résultat sera :

Nom	Valeur
a	1
b	2

ou

Nom	Valeur
a	2
b	1

En effet, un dictionnaire est utilisé pour passer les arguments. Or ceux-ci ne garantissent pas d'ordre sur leurs clés. Pourtant, Python dispose d'une classe `OrderedDict` depuis Python 3.1. L'implémentation de cette PEP se résumerait donc à remplacer l'objet utilisé en interne, un dictionnaire basique, par un dictionnaire ordonné. Cependant, jusqu'à maintenant, cet objet était défini en Python. L'implémentation était donc loin d'être la plus efficace possible et ne pouvait pas être utilisée pour un élément aussi critique du langage. C'est pour cette raison que la classe a été réimplémentée en C pour Python 3.5, comme noté dans la section « De plus petits changements ». Maintenant, plus rien ne semble bloquer l'implémentation de cette PEP, qui devrait donc voir le jour dans Python 3.6.

Le principal intérêt de cette modification est que le fonctionnement actuel est contre-intuitif pour bon nombre d'utilisateurs connaissant mal le fonctionnement interne de Python. D'autres raisons sont invoquées dans la PEP :

— L'initialisation d'un `OrderedDict` est un cas d'appel où l'ordre des paramètres importe, et de tels objets pourraient maintenant être créés en utilisant des arguments nommés, comme le permettent les dictionnaires.
— La sérialisation : dans certains formats l'ordre d'apparition des données a de l'importance (ex : l'ordre des colonnes dans un fichier CSV). Cette nouvelle possibilité permettrait de les définir plus facilement, en même temps que des valeurs par défaut. Elle permettrait aussi à des formats comme XML, JSON ou Yaml de garantir l'ordre d'apparition des attributs ou

clés qui sont enregistrés dans les fichiers.
— Le débogage : le fonctionnement actuel pouvant être aléatoire, il peut être compliqué de reproduire certains bugs. Si un bug apparait selon l'ordre de définition des arguments, le nouveau comportement facilitera leur correction.
— La priorité à donner aux arguments pourrait être spécifiée selon l'ordre de leur déclaration.
— Les `namedtuple` pourraient être définis aisément avec une valeur par défaut.

Et probablement d'autres utilisations qui n'ont pas encore été envisagées.

5.3 Proprités de classes

Toujours dans les petites modifications, un nouveau décorateur disponible de base devrait être introduit : `@classproperty`. Si vous connaissez le modèle objet de Python, son nom devrait vous suffir pour deviner son but : permettre de définir des propriétés au niveau de classes. Par exemple si vous souhaitez conserver au niveau de la classe le nombre d'instances créées et des statistiques sur vos appels :

```
class Spam:
    _n_instance_created = 0
    _n_egg_call = 0

    def __init__(self):
        self.__class__._n_instance_created += 1

    def egg(self):
        print("egg")
        self.__class__._n_egg_call += 1

    @classproperty
    def mean_egg_call(cls):
        return (cls._n_egg_call / cls._n_instance_created) if cls._n_instance_created

spam = Spam()

spam.egg()
spam.egg()
spam.egg()

print(spam.mean_egg_call)   # => 3

spam2 = Spam()

## Les 3 expressions suivantes sont équivalentes
print(spam.mean_egg_call)    # => 1.5
print(spam2.mean_egg_call)   # => 1.5
print(Spam.mean_egg_call)    # => 1.5  , propriété au niveau de la classe !
```

Cet ajout a été proposé sur la *mailling-list python-ideas*. La mise en place d'une implémentation propre et complète de ce comportement est compliquée sans définir une méta-classe. Une solu-

tion générique implique donc de le rajouter dans le modèle objet de Python. Guido a rapidement donné son aprobation et un ticket a été créé en attendant qu'un des développeurs de CPython l'implémente dans le coeur de l'interpréteur. Cela pourrait donc être l'une des premières nouvelles fonctionnalités confirmées pour la version 3.6.

5.4 Sous-interpréteurs

[[a]] | Cette section peut nécessiter des notions avancées de programmation système pour être comprise, en particulier la différence entre un *thread* et un *processus*, ainsi que leur gestion dans Python.

L'écriture de code concurrent en Python est toujours un sujet chaud. À ce jour, trois solutions existent dans la bibliotèque standard :

— *asyncio*, bien que limitée à un seul *thread*, permet d'écrire des coroutines s'exécutant en concurrence en tirant partie des temps d'attente introduits par les entrées/sorties.
— *threading* permet de facilement exécuter du code sur plusieurs *threads*, mais leur exécution reste limitée à un seul coeur de processeur à cause du GIL.
— *multiprocessing*, en *forkant* l'interpréteur, permet d'éxecuter plusieurs codes Python en parallèle sans limitation et exploitant pleinnement les ressources calculatoires des processeurs.

[[a]] | Le GIL est une construction implémentée dans de nompreux interpéteurs (CPython, Pypy, Ruby, etc.). Ce mécanisme bloque l'interpréteur pour qu'à chaque instant, un seul code puisse être exécuté. Ce sytème permet de s'assurer que du code éxécuté sur plusieurs *threads* ne va pas poser de problèmes de concurrence sur la mémoire, sans vraiment ralentir les codes n'utilisant qu'un seul *thread*. Malheureusement cela nous empèche d'exploiter les architectures multi-coeurs de nos processeurs.

La multiplicité des solutions ne résoud pas tout. En effet les limitation des *threads* en Python les rendent inutiles quand le traitement exploite principalement le processeur. L'utilisation de *multiprocessing* est alors possible, mais a un coût :

— Le lancement d'un processus entraine un *fork* au niveau du système d'exploitation, ce qui prend plus de temps et de mémoire que le lancement d'un *thread* (presque gratuit en comparaison).
— La communication entre les processus est aussi plus longue. Tandis que les *threads* permettent d'exploiter un espace partagé en mémoire, rendant leurs communications directes, les processus nécessitent de mettre en place des mécanismes complexes, appelés *IPC*.

*[IPC] : Communication inter-processus

Une proposition sur *python-ideas* a été formulée au début de l'été pour offrir une nouvelle solution intermédiare entre les *threads* et les processus : des sous-interpréteurs (*subinterpreters*), en espérant que cela puisse définitivement contenter les développeurs friands de codes concurrents.

À partir d'un nouveau module *subinterpreters*, reprenant l'interface exposée par les modules *threading* et *multiprocessing*, CPython permettrait de lancer dans des *threads* différents interpréteurs, chacun chargé d'executer un code Python. Le fait que ce soient des *threads* rendrait ce module beaucoup plus léger à utiliser que *multiprocessing*. L'interpréteur se chargerait de lancer ces *threads* et partagerait le maximum de son implémentation entre eux. Le plus intéressant est que ce mécanisme permettrait d'éluder le *GIL*. En effet, chaque sous-interpréteur disposerait d'un espace

de noms propre et indépendant. Chacun aurait donc un *GIL*, mais ceux-ci seraient indépendants. D'un point de l'ensemble, la majorité des opérations transformeraient le *GIL* en *LIL* : *Local interpreter lock*. Chaque sous-interpréteur pourrait ainsi fonctionner sur un coeur du processus séparé sans aucun problème, permettant ainsi au développeur de tirer pleinement partie des processeurs modernes. Enfin, il faut savoir que CPython possède déjà en interne la notion de sous-interpréteurs, le travail à réaliser consisterait donc à exposer ce code C pour le rendre disponible en Python.

Évidement, cette proposition n'est pas une solution miracle. Tout n'est pas si simple et beaucoup d'élements doivent encore être discutés. En particulier les moyens de communication entre les sous-interpréteurs (probablement via des queues, comme pour *multiprocessing*) et tout un tas de petits détails d'implémentation. Mais la proposition a reçu un accueil très positif et l'auteur est actuellement en train de préparer une PEP à ce sujet.

[GIL] : Global Interpreter Lock*

5.5 Interpolation de chaines

Les interpolations directes de chaînes de caractères existent dans de nombreux langages : PHP, C#, Ruby, Swift, Perl, etc. En voici un exemple en Perl :

```
my $a = 1;
my $b = 2;

print "Resultat = a + b = $a + $b = @{[$a+$b]}\n";
## Imprime "Resultat = a + b = 1 + 2 = 3"
```

Il n'existe pas d'équivalent direct en Python ; le plus proche pourrait ressembler à l'exemple suivant :

```
a, b = 1, 2

print("Resultat = a + b = %d + %d = %d" % (a, b, a + b))
## ou
print("Resultat = a + b = {a} + {b} = {c}".format(a=a, b=b, c=a + b))
```

Nous voyons ainsi qu'il est nécessaire de passer explicitement les variables et, même s'il est possible d'utiliser `locals()` ou `globals()` pour s'en passer, il n'est possible d'évaluer une expression, comme ici l'addition, qu'à l'extérieur de la chaîne de caractères puis d'injecter le résultat dans cette dernière.

La première proposition effectuée, formalisée par la PEP 498, propose de rajouter cette possibilité dans Python grâce à un nouveau préfixe de chaîne, `f`, pour `format-string`, dont voici un exemple d'utilisation, issu de la PEP :

```
>>> import datetime
>>> name = 'Fred'
>>> age = 50
>>> anniversary = datetime.date(1991, 10, 12)
>>> f'My name is {name}, my age next year is {age+1}, my anniversary is {anniversary:%
'My name is Fred, my age next year is 51, my anniversary is Saturday, October 12, 1991
>>> f'He said his name is {name!r}.'
"He said his name is 'Fred'."
```

Cette proposition a fait des émules. Plusieurs développeurs ont mit en évidence que la méthode d'interpolation peut dépendre du contexte. Par exemple un module de base de données comme `sqlite3` propose un système de substitution personnalisé pour éviter les attaques par injection ; les moteurs de *templates*, eux, pour produire du HTML, vont aussi faire des substitutions pour échapper certains caractères, comme remplacer < ou > par, respectivement, `<` ou `>`. La PEP 501 propose donc aux développeurs de définir leurs propres méthodes d'interpolation, adaptées au contexte.

Guido a déjà accepté la première proposition afin qu'elle soit implémentée dans Python 3.6. La deuxième est plus générale mais plus complexe, et peut ainsi poser plusieurs problèmes. Aucune décision n'a encore été prise et les débats à ce sujet ont encore lieu sur la *mailling list* concernant cette possibilité et la forme qu'elle pourrait prendre. Le *patch* pour la PEP 498 accepté est presque prêt et sera donc vraisemblablement la première nouvelle fonctionnalité de Python 3.6 ; l'implémentation de la PEP 501 dépendra de la suite des discussions.

6 Conclusion

Vous en conviendrez, cette version est sans doute l'une des plus importantes en termes de nouvelles fonctionnalités. Qui plus est, aujourd'hui, plus aucune fonctionnalité n'est introduite dans Python 2.7 et la plupart des bibliothèques populaires sont compatibles avec Python 3, nous pouvons espérer que cette version permette enfin de parachever le basculement de Python 2 à Python 3.

Vous pouvez dès maintenant télécharger cette nouvelle version sur le site officiel de Python.

Enfin, si certains points de cet article ne sont pas clairs pour vous, n'hésitez pas à nous en faire part en commentaire et nous nous efforcerons de clarifier vos interrogations !