ENTWANNE

Zeste de savoir

Sortie de Python 3.6

12 août 2019

Table des matières

Nous vous parlions il y a un an de la sortie de la version 3.5 du langage Python . Cette version portait notamment sur la programmation asynchrone (avec les mots-clés `async` et `await`) et l'*unpacking* (opérateurs *splat* : `*` et `**`).

Les mois ont passé et arrivent aujourd'hui Python 3.6 (nouvelle version du langage) et CPython 3.6 (interpréteur officiel écrit en C). Cette version ajoute au langage l'interpolation de chaînes, les arguments nommés ordonnés, et quelques subtilités sur la création de classes. Des fonctionnalités secondaires, mais attendues depuis un certain temps par les développeurs Python.

Cet article a pour but de faire le tour de cette nouvelle version et de vous présenter les changements qu'elle apporte.

1. TL ;DR - Résumé des principales nouveautés

Comme pour la version précédente, commençons par un résumé des principales nouveautés . Les fonctionnalités listées ici seront bien sûr détaillées par la suite. Pour rappel, les PEP sont des spécifications décrivant les potentielles fonctionnalités du langage.

— PEP 498 : les chaînes de caractères peuvent maintenant être préfixées du symbole `f` pour être interpolées en fonction des variables du *scope* courant.
Cette fonctionnalité reprend globalement la syntaxe supportée par la méthode `format` des chaînes de caractères.

```
>>> name = 'John'
>>> f'Hello {name}!'
'Hello John!'
```

— PEP 468 : les arguments nommés reçus par une fonction sont maintenant assurés d'être ordonnés. Le paramètre spécial `**kwargs` d'une fonction correspondra alors toujours à un dictionnaire ordonné des arguments nommés.

```
>>> def func(**kwargs):
...     for name, value in kwargs.items():
...         print(name, '->', value)
...
>>> func(a=1, b=2, c=3)
a -> 1
b -> 2
c -> 3
>>> func(b=2, c=3, a=1)
b -> 2
c -> 3
a -> 1
```

— PEP 519 : ajout d'un protocole pour les chemins de fichiers. Il devient maintenant plus facile de manipuler des chemins et d'interagir avec les fonctions système.

```
>>> import pathlib
>>> path = pathlib.Path('/home')
>>> path / 'john'
PosixPath('/home/john')
>>> with open(path / 'john' / '.python_history') as f:
...     history = f.read()
...
```

— PEP 520 : le dictionnaire de définition d'une classe devient lui aussi ordonné. L'ordre de définition des attributs et méthodes des classes est donc conservé.

```
>>> class A:
...     x = 0
...     y = 0
...     z = 0
...
>>> A.__dict__
mappingproxy({..., 'x': 0, 'y': 0, 'z': 0, ...})
```

— PEP 487 : une nouvelle méthode (`__init_subclass__`) permet d'interférer depuis une classe mère sur la création de ses filles ; et les descripteurs sont informés lorsqu'ils sont assignés à un attribut de la classe, *via* leur méthode `__set_name__`.

— PEP 525 : il devient possible d'écrire des générateurs asynchrones. Ils fonctionneront

comme des générateurs habituels, à utiliser depuis d'autres coroutines.

```
async def async_range(start, stop):
    for i in range(start, stop):
        yield i
```

— PEP 530 : le mot-clé `async` peut maintenant être utilisé dans les listes en intension (et autres compréhensions) au sein d'une coroutine.

```
async def coroutine():
    return [i async for i in async_range(0, 10)]
```

2. Principales nouveautés

2.1. Interpolation de chaînes littérales — PEP 498

Loin sont maintenant les `'My name is {name}'.format(name=name)` ou encore `'My name is {}'.format(name)`. Avec Python 3.6, le formatage de chaînes de caractères gagne en clarté avec l'interpolation littérale.

En plus du préfixe `r` pour définir une chaîne brute, le préfixe `b` pour une chaîne d'octets, arrive maintenant le préfixe `f`, dédié au formatage. Une chaîne préfixée de `f` sera interpolée à sa création, de manière à résoudre les expressions utilisées au sein de la chaîne de formatage.

La syntaxe pour l'interpolation littérale est reprise sur la syntaxe de la méthode `str.format` . Au détail près que sont accessibles les variables de l'espace de nom courant plutôt que seulement les valeurs qui étaient auparavant passées en arguments.

```
>>> name = 'John'
>>> f'Hello {name}'
'Hello John'
>>> # Équivalent à
>>> 'Hello {name}'.format(name=name)
'Hello John'
>>> 'Hello {name}'.format(**locals())
'Hello John'
```

Des formatages plus complets sont permis par la méthode `__format__` de certains objets, comme ici pour ceux de type `datetime.date` .

```
>>> import datetime
>>> date = datetime.date.today()
```

```
>>> f'Cet article a été écrit le {date}'
'Cet article a été écrit le 2016-10-27'
>>> f'Cet article a été écrit le {date:%d %B %Y}'
'Cet article a été écrit le 27 octobre 2016'
```

Les accolades peuvent non seulement contenir un nom de variable, mais aussi toute expression Python valide.

```
>>> nb_pommes = 5
>>> print(f"J'ai {nb_pommes} pomme{'s' if nb_pommes > 1 else ''}.")
J'ai 5 pommes.
>>> nb_pommes = 1
>>> print(f"J'ai {nb_pommes} pomme{'s' if nb_pommes > 1 else ''}.")
J'ai 1 pomme.
```

2.2. Préservation de l'ordre des arguments nommés — PEP 468

Les paramètres du type `**kwargs` dans la signature d'une fonction sont maintenant assurés d'être des dictionnaires ordonnés. Les arguments nommés sont ainsi récupérés dans l'ordre où ils ont été saisis.

```
def xml_tag(name, **kwargs):
    lines = [f'<{name}>']
    for key, value in kwargs.items():
        lines.append(f'    <{key}>{value}</{key}>')
    lines.append(f'</{name}>')
    return '\n'.join(lines)
```

Cette fonction nous permet de sérialiser un extrait XML tout en conservant le bon ordre des éléments fils.

```
>>> print(xml_tag('book', title='Notre-Dame de Paris',
    author='Victor Hugo', year=1831))
<book>
    <title>Notre-Dame de Paris</title>
    <author>Victor Hugo</author>
    <year>1831</year>
</book>
```

Il devient aussi possible de construire facilement un objet `OrderedDict`.

```
from collections import OrderedDict
coords = OrderedDict(x=0, y=5, z=1)
```

Là où une liste de tuples était nécessaire dans les versions antérieures de Python.

```
coords = OrderedDict([('x', 0), ('y', 5), ('z', 1)])
```

On notera que cela passe, dans CPython 3.6, par une implémentation ordonnée de tous les `dict`.

```
>>> {'x': 0, 'y': 5, 'z': 1}
{'x': 0, 'y': 5, 'z': 1}
```

CPython reprend ici les travaux entrepris par *pypy* pour construire une version des dictionnaires plus compacte, c'est-à-dire occupant moins d'espace en mémoire.

2.3. Protocole de gestion des chemins de fichiers — PEP 519

Cette PEP revient sur la `pathlib`, bibliothèque de gestion des chemins de fichiers, pour lui ajouter son propre protocole.

Spécialisée dans la manipulation de chemins de fichiers, cette bibliothèque comporte aussi de nombreux outils pour opérer sur ces fichiers (création de fichier, de dossier, suppression, etc.)

Une nouvelle méthode spéciale est maintenant disponible pour nos objets : la méthode `__fspath__`. Elle ne prend aucun paramètre et doit retourner une chaîne de caractères. Tout objet implémentant cette méthode est considéré par Python comme un chemin, et peut alors être utilisé avec les fonctions Python gérant des chemins (`open`, `os.path.join`, etc.).

```
import os

class UserHome:
    def __init__(self, username):
        self.username = username

    def __fspath__(self):
        return f'/home/{self.username}'

for filename in os.listdir(UserHome('clem')):
    print(filename)
```

Nous avons ici notre propre type d'objet (`UserHome`), qui est interprété par les fonctions système (`os.listdir` dans notre cas) comme un chemin de fichier.

2.4. Préservation de l'ordre des attributs définis dans les classes — PEP 520

Similairement à la PEP 468, le dictionnaire des attributs d'une classe est maintenant assuré d'être ordonné. Il conservera alors l'ordre de définition des attributs et méthodes dans le corps de la classe.

Cela peut servir pour des classes dont l'ordre des attributs/méthodes serait important, telle qu'une énumération (`Enum`). Cela permet aussi une meilleure introspection des classes.

```
class AutoEnum:
    red = None
    green = None
    blue = None

i = 0
for name, value in AutoEnum.__dict__.items():
    if not name.startswith('__') and not name.endswith('__') and
        value is None:
        setattr(AutoEnum, name, i)
        i += 1
```

```
>>> print(AutoEnum.__dict__)
{'__module__': '__main__', 'red': 0, 'green': 1, 'blue': 2, ...}
```

Il était autrefois possible d'avoir un dictionnaire ordonné pour les attributs en passant par une métaclasse. En effet, la méthode `__prepare__` des métaclasses permet de spécialiser la création du dictionnaire `__dict__`, et donc de retourner un `OrderedDict` si besoin.

La volonté de cette PEP (et de la 487) est de réduire le besoin de recourir aux métaclasses pour des problèmes simples.

2.5. Simplification de la personnalisation de classes — PEP 487

2.5.1. Héritage

Une classe peut maintenant influer sur les classes qui en héritent. C'est ce que permet la méthode `__init_subclass__` nouvellement ajoutée par cette PEP. La méthode sera appelée chaque fois qu'une classe fille sera créée (et non instanciée), et la classe fille passée en paramètre.

```
>>> class SubclassMePlease:
...     def __init_subclass__(cls):
...         print('Creation of', cls)
...
>>> class Ok(SubclassMePlease):
...     pass
...
Creation of <class '__main__.Ok'>
```

Cette méthode de classe reçoit aussi en paramètre l'ensemble des arguments nommés passés lors de l'héritage. Vous ne le saviez peut-être pas, mais des arguments peuvent être donnés lors de la création d'une classe (notamment pour la précision de la métaclasse). Il était déjà possible auparavant de les récupérer dans les méthodes `__new__` et `__init__` des métaclasses.

```
>>> class SubclassMePlease:
...     def __init_subclass__(cls, *, name, **kwargs):
...         print('Creation of', cls, 'with name', name)
...         cls.name = name
...
>>> class Roger(SubclassMePlease, name='roger'):
...     pass
...
Creation of <class '__main__.Roger'> with name roger
>>> Roger.name
'roger'
```

2.5.2. Descripteurs

Cette PEP ajoute aussi une nouvelle méthode spéciale aux descripteurs , la méthode `__set_name__`. Elle permet au descripteur de savoir quel nom d'attribut lui a été donné au sein de la classe. En effet, la méthode sera appelée suite à la création de la classe, avec comme arguments la classe et le nom du descripteur.

Cela peut s'avérer utile pour les descripteurs qui donnent accès à un autre attribut de l'objet, dont le nom peut maintenant être interpolé depuis celui du descripteur.

Dans l'exemple suivant, nous avons des descripteurs `width` et `height` qui permettent un accès en lecture à `_width` et `_height`.

```
class AttributeReader:
    def __set_name__(self, owner, name):
        self.attr = '_{}'.format(name)

    def __get__(self, instance, owner):
```

```
6           if instance is None:
7               return self
8           return getattr(instance, self.attr)
9
10  class Rect:
11      width = AttributeReader()
12      height = AttributeReader()
13
14      def __init__(self, width, height):
15          self._width, self._height = width, height
```

2.6. Générateurs et compréhensions asynchrones — PEP 525 & PEP 530

Les paragraphes qui suivent demandent des connaissances sur les concepts de générateurs et de programmation asynchrone.

Les mots-clés `async` et `await` introduits avec Python 3.5 connaissent une nouvelle extension. En effet, il devient désormais possible de coupler les coroutines avec des générateurs et des compréhensions (listes en intension, *generator expressions*).

C'est-à-dire que l'on va pouvoir définir un générateur asynchrone, qui dépendra d'événements externes pour produire ses valeurs. L'itération sur ces générateurs devra alors être réalisée *via* `async for` plutôt qu'un simple `for`, au sein de coroutines.

Imaginons un générateur qui produirait les lignes d'un texte en les récupérant depuis un programme distant. Nous le représenterons ici par une itération sur un fichier, faisant une pause d'une seconde après chaque ligne pour illustrer une certaine latence.

```
1  import asyncio
2
3  async def produce_lines(filename):
4      with open(filename, 'r') as f:
5          for line in f:
6              yield line.rstrip('\n')
7              await asyncio.sleep(1)
```

On reconnaît que `produce_lines` est un générateur à l'utilisation du mot-clé `yield` en ligne 6.

Nous pouvons alors avoir une seconde coroutine, `print_lines`, qui itèrera sur ce générateur pour afficher les lignes produites.

```
async def print_lines(filename):
    async for line in produce_lines(filename):
        print(f'[{filename}]', line)
```

Et à l'utilisation :

```
>>> loop = asyncio.get_event_loop()
>>> loop.run_until_complete(print_lines('corbeau.txt'))
[corbeau.txt] Maître Corbeau, sur un arbre perché,
[corbeau.txt] Tenait en son bec un fromage.
[corbeau.txt] Maître Renard, par l'odeur alléché,
[corbeau.txt] Lui tint à peu près ce langage :
[...]
>>>
    loop.run_until_complete(asyncio.wait([print_lines('corbeau.txt'),
...
    print_lines('loup.txt')]))
[corbeau.txt] Maître Corbeau, sur un arbre perché,
[loup.txt] La raison du plus fort est toujours la meilleure :
[corbeau.txt] Tenait en son bec un fromage.
[loup.txt] Nous l'allons montrer tout à l'heure.
[corbeau.txt] Maître Renard, par l'odeur alléché,
[loup.txt] Un Agneau se désaltérait
[corbeau.txt] Lui tint à peu près ce langage :
[loup.txt] Dans le courant d'une onde pure.
[...]
```

Quant aux compréhensions asynchrones, introduites par la PEP 530, elles s'illustrent par la coroutine suivante, chargée de retourner la liste de ces lignes.

```
async def get_lines(filename):
    return [line async for line in produce_lines(filename)]
```

Notez bien le `async for` utilisé dans la compréhension.

L'appel à notre coroutine au sein de notre boucle événementielle nous retourne donc ici la liste des lignes.

```
>>> loop.run_until_complete(get_lines('corbeau.txt'))
['Maître Corbeau, sur un arbre perché,',
    'Tenait en son bec un fromage.', ...]
```

3. De plus petits changements

3.0.1. PEP 526 : annotations de variables.

Après les paramètres de fonctions, ce sont maintenant toutes les variables qui peuvent être annotées. Le même principe est conservé : les annotations n'ont aucune utilité dans l'interpréteur Python, mais sont stockées dans l'attribut `__annotations__` du *scope* parent (classe, module). Les annotations servent pour les outils externes d'analyse statique du code par exemple (`mypy`).

```
>>> a : int
>>> b : str = 'hello'
>>> a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'a' is not defined
>>> b
'hello'
>>> __annotations__
{'a': <class 'int'>, 'b': <class 'str'>}
```

```
>>> class A:
...     x : int = 0
...     y : int = 0
...     name : str
...
>>> A.__annotations__
{'x': <class 'int'>, 'y': <class 'int'>, 'name': <class 'str'>}
```

3.0.2. PEP 515 : *underscores* dans les nombres littéraux.

Les grands nombres compacts ne sont pas toujours aisés à lire. Ainsi, on mettra du temps à déceler l'ordre de grandeur dans `1000000000`. Des *underscores* peuvent maintenant être utilisés dans l'écriture littérale des nombres, pour les séparer en différents blocs de chiffres : `1_000_000_000`.

```
>>> 1_000_000 * 1_000
1000000000
>>> 0b_11001100_00000001
52225
```

— PEP 506 : ajout d'un module `secrets` pour générer des nombres aléatoires plus sûrs.

— PEP 523 : ajout d'une API pour l'évaluation de *frames* de CPython.
— PEP 528 et PEP 529 : utilisation d'UTF-8 pour la console et les chemins de fichiers sur Windows.
— PEP 509 et PEP 510 : outils d'optimisation au cœur de l'interpréteur (dictionnaires versionnés et *guards* sur les fonctions).
— PEP 495 : désambiguïsation des temps identiques après changement d'heure.
— PEP 524 : la fonction `os.urandom` est maintenant bloquante sur les systèmes Linux pour une sécurité accrue.

Cette version 3.6 vient aussi avec son lot de corrections de bogues, que vous pourrez retrouver dans le *changelog* de la version.

4. Ce que l'on peut attendre pour la version 3.7

Personne ne les décidant par avance, les fonctionnalités de la future version 3.7 ne sont pas encore connues. Néanmoins, ces fonctionnalités découlent des propositions faites par les utilisateurs/développeurs, généralement sur l'une de ces deux *mailing lists* :

— *python-dev* pour les changements concernant l'intepréteur CPython ;
— *python-ideas* pour les idées générales sur le langage.

Ces idées sont ensuite débattues, et mènent à une PEP si elles sont acceptées par une majorité de développeurs. La PEP 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.

> *i* Partant de ces propositions, il est possible d'établir une liste de possibles futures fonctionnalités.

Les **propriétés de classes** et des **sous-interpréteurs**, abordés dans le précédent article , ne seront pas à décrites ici.

4.1. Syntaxe pour les arguments uniquement positionnels (PEP 457)

Les arguments positionnels en Python sont les arguments passés aux fonctions qui sont assignés aux paramètres suivant leur position. Par exemple, avec une fonction `def add(x, y): pass`, dans l'expression `add(3, 5)`, le paramètre `x` récupérera la valeur du premier argument (3), et `y` celle du second (5). Avec une expression telle que `add(x=3, y=5)` (ou `add(y=5, x=3)` équivalente), la correspondance se fait par le nom des paramètres, on parle alors d'arguments nommés.

Les paramètres d'une fonction en Python peuvent alors correspondre à des arguments positionnels ou des arguments nommés, il est aussi possible de faire en sorte qu'ils ne puissent être définis que *via* des arguments nommés (avec une fonction telle que `def add(*, x, y)` par exemple).

Néanmoins, bien que prévus par l'implémentation (la fonction `pow` ne reçoit que des arguments positionnels), il n'existe aucune syntaxe en Python permettant d'avoir des arguments uniquement

positionnels. Il est ici proposé de pouvoir ajouter un caractère `/` lors de la définition des paramètres, lequel serait précédé des arguments uniquement positionnels.

```
def add(x, y, /):
    return x + y
```

4.2. Expressions attrapant les exceptions (PEP 463)

Cette PEP vise à permettre d'attraper des exceptions sous forme d'expressions, évitant de créer un bloc `try/except` dans les cas les plus simples.

L'idée serait d'avoir une expression `expression except exception_type: default_value` qui évaluerait l'expression `expression` et prendrait sa valeur, mais dans le cas où `expression` lèverait une exception de type `exception_type`, le résultat de l'expression serait `default_value`.

Ainsi, les deux codes suivants seraient équivalents.

```
result = a / b except ZeroDivisionError: 0
```

```
try:
    result = a / b
except ZeroDivisionError:
    result = 0
```

4.3. Généralisation de l'interpolation de chaînes (PEP 501)

L'interpolation de chaînes arrivée avec Python 3.6 ouvre de nouvelles perspectives pour les versions suivantes.

À l'instar du nouveau préfixe `f` pour les chaînes de caractères, cette PEP demande l'ajout d'un préfixe `i` tel que `f'Bonjour {name}'` soit équivalent à `format(i'Bonjour {name}')`. L'expression `i'...'` retournerait un objet d'un nouveau type (un *template*) dont la méthode `__format__` (appelée par `format(...)`) se chargerait du formatage proprement dit, correspondant à l'actuel préfixe `f`.

Cette PEP permettrait d'appliquer plusieurs fois `format` sur un *template* et donc de le réutiliser. Il deviendrait aussi possible d'étendre le type de *template* pour implémenter sa propre méthode `__format__` et bénéficier d'un formatage différent (pour des raisons de sécurité par exemple).

4.4. Opérateurs de coalescence (PEP 505)

Les opérateurs de coalescence (*null-coalescing*) sont des opérateurs qui simplifient la gestion des valeurs nulles, que l'on retrouve dans plusieurs langages (C#, Ruby, Perl, PHP, etc.). Ils permettent de n'exécuter la suite d'une expression que si la valeur utilisée n'est pas nulle.

Le concept pourrait être adapté en Python autour de la valeur `None`. Il est courant de rencontrer des expressions telles que `obj.attr if obj is not None else 'default'` pour récupérer l'attribut `attr` d'un objet `obj` incertain (pouvant être un objet du type désiré ou `None`). Cette expression présente le problème de s'avérer un peu lourde et d'être répétitive (`obj` y apparaît deux fois). Si `obj` était une expression plus complète, on pourrait aussi avoir un soucis de double-évaluation.

Les opérateurs de coalescence seraient au nombre de 3 :

— `??` (*None-coalescing*), afin d'avoir une valeur par défaut si l'expression à gauche de l'opérateur s'évalue à `None` ;
— `?.` (*None-aware attribute access*), pour n'accéder à l'attribut d'un objet que si cet objet n'est pas `None` (comme dans l'exemple plus haut) ;
— `?.[]` (*None-aware indexing*) permet de faire de même lors de l'accès aux éléments d'un conteneur (`container[...]`).

On aurait alors :

```
>>> number = 5
>>> number ?? 100
5
>>> number = None
>>> number ?? 100
100
>>> number = 0
>>> number ?? 100
0

>>> obj = {'a': 5}
>>> obj?.popitem('a')
5
>>> obj = None
>>> obj?.popitem('a')
None

>>> obj = {'a': 5}
>>> obj?.['a']
5
>>> obj = None
>>> obj?.['a']
None
```

4. Ce que l'on peut attendre pour la version 3.7

Python 3.6 ne bouscule donc pas l'écosystème du langage en n'apportant que quelques modifications cosmétiques. Ces modifications étaient toutefois attendues de longue date, et améliorent encore le confort des développeurs.

Cette nouvelle version est disponible sur le site officiel de Python, à la page des téléchargements
.

Pour toute question/remarque sur Python 3.6 ou sur le présent article, la section commentaires est à votre libre disposition. Je me ferai un plaisir de répondre à vos interrogations !

Liste des abréviations

BDFL Benevolent Dictator for Life (Dictateur bienveillant à vie). 11

PEP Python Enhancement Proposal (Proposition d'amélioration de Python). 1–13