ENTWANNE

Zeste de savoir

Sortie de Python 3.9

5 octobre 2020

Table des matières

Le 5 octobre 2020 est sortie la version 3.9 du langage de programmation Python .

Cette version marque une profonde refonte de l'interpréteur Python dans la manière d'analyser le code (voir section «Refonte de l'analyseur syntaxique ») mais peu de changements en surface.
Elle est aussi la première version du nouveau cycle de sorties : auparavant une nouvelle version de Python sortait tous les 18 mois, cette durée est maintenant ramenée à un an.

On notera que contrairement à une idée répandue il ne s'agit pas de la dernière des 3.x : il y aura une version 3.10 et la 4.0 n'est pas pour tout de suite. Cette dernière n'est cependant pas à redouter, le passage de Python 3 à Python 4 sera bien plus transparent que celui de 2 à 3 (il s'apparentera plus au passage de Python 1 à Python 2).

Mais pour le moment, voici un rapide tour d'horizon des principales fonctionnalités de Python 3.9.

1. Méthodes removeprefix et removesuffix des chaînes de caractères

La suppression d'un préfixe ou d'un suffixe depuis une chaîne de caractères est une opération relativement courante.

Par exemple, on aimerait pouvoir supprimer le préfixe `'Zeste'` depuis la chaîne `'ZesteDeSavoir'` pour isoler la partie `'DeSavoir'`. De la même manière, `'ZesteDeCitron'` deviendrait `'DeCitron'`.

Avant Python 3.9, on aurait pu pour cela procéder avec du *slicing* .

```
>>> 'ZesteDeSavoir'[5:]
'DeSavoir'
```

```
>>> 'ZesteDeCitron'[5:]
'DeCitron'
```

Deux problèmes à cela :

— Il faut indiquer la taille du préfixe ce qui n'est pas très explicite, ça pourrait se régler en écrivant `'ZesteDeSavoir'[len('Zeste'):]` mais ça devient un peu lourd.
— Cela fonctionne si le préfixe existe bien dans la chaîne, mais donne des résultats incohérents sinon.

```
>>> 'SmoothieAuCitron'[5:]
'hieAuCitron'
```

Dans le cas où le préfixe n'existe pas, on aimerait que la chaîne reste inchangée. C'était encore une fois possible de passer outre ce problème en utilisant au préalable un test `startswith`.

```
>>> prefix = 'Zeste'
>>> string = 'ZesteDeCitron'
>>> string[len(prefix):] if string.startswith(prefix) else string
'DeCitron'
>>> string = 'SmoothieAuCitron'
>>> string[len(prefix):] if string.startswith(prefix) else string
'SmoothieAuCitron'
```

Ce qui donne une expression relativement complexe pour une opération plutôt simple. Et c'est là qu'arrive la PEP 616 de Python 3.9 qui fournit une méthode `removeprefix` pour appliquer simplement cette opération.

```
>>> 'ZesteDeSavoir'.removeprefix('Zeste')
'DeSavoir'
>>> 'ZesteDeCitron'.removeprefix('Zeste')
'DeCitron'
>>> 'SmoothieAuCitron'.removeprefix('Zeste')
'SmoothieAuCitron'
```

Est apportée aussi la méthode `removesuffix`, l'opération analogue pour enlever les suffixes.

```
>>> 'ZesteDeSavoir'.removesuffix('Savoir')
'ZesteDe'
>>> 'ZesteDeCitron'.removesuffix('Savoir')
'ZesteDeCitron'
```

2. Opérations ensemblistes sur les dictionnaires

La PEP 584 apporte un nouvel opérateur sur les dictionnaires, l'opérateur d'union (|). Celui-ci permet de fusionner deux dictionnaires, comme il le fait pour les ensembles.

```
>>> d1 = {'steak': 1, 'frites': 2}
>>> d2 = {'boisson': 1}
>>> d1 | d2
{'steak': 1, 'frites': 2, 'boisson': 1}
```

En cas de clé commune, c'est celle du dictionnaire de droite qui a la priorité.

```
>>> {'foo': 0, 'bar': 0} | {'foo': 1, 'baz': 1}
{'foo': 1, 'bar': 0, 'baz': 1}
```

Auparavant il était possible de réaliser cette opération avec un `{**d1, **d2}` mais cette expression n'était pas claire pour tout le monde.

Vient aussi avec cette même PEP l'opérateur `|=` pour faire une fusion en place sur un dictionnaire, équivalent à la méthode `update`.

```
>>> d1 |= d2
>>> d1
{'steak': 1, 'frites': 2, 'boisson': 1}
```

3. Module zoneinfo

Un nouveau module fait son apparition en Python 3.9, le module `zoneinfo` . Ce module permet d'ajouter dans le cœur de Python le support de divers fuseaux horaires (base IANA) via une classe `ZoneInfo`.

Ces fuseaux correspondent aux différentes zones géographiques terrestres, comme `Europe/Paris` ou `America/Montreal`, et permettent une gestion haut-niveau du fuseau sans avoir à connaître le décalage par rapport à UTC ou à gérer les heures d'été.

```
>>> from datetime import datetime
>>> from zoneinfo import ZoneInfo
>>> dt = datetime(2020, 2, 29, 12, 0,
    tzinfo=ZoneInfo('Europe/Paris')) # Le 29/02/2020 à 12:00 à
    Paris
>>> print(dt)
```

```
2020-02-29 12:00:00+01:00
>>> print(dt.astimezone(ZoneInfo('UTC'))) # Conversion UTC
2020-02-29 11:00:00+00:00
>>> dt = datetime(2020, 5, 17, 12, 0,
    tzinfo=ZoneInfo('Europe/Paris')) # Le 17/05/2020 à 12:00 à
    Paris
>>> print(dt)
2020-05-17 12:00:00+02:00
>>> print(dt.astimezone(ZoneInfo('UTC'))) # Conversion UTC
2020-05-17 10:00:00+00:00
```

Plus d'informations à ce sujet dans la PEP 615 .

4. Module graphlib

Python 3.9 apporte un autre nouveau module à la biliothèque standard : `graphlib` .
C'est un module fournissant une classe `TopologicalSorter` pour réaliser des tris topologiques sur des graphes.

La documentation du module décrit cela plus en détails, mais ce tri permet d'obtenir en premier les nœuds les plus bas dans la hiérarchie (ceux qui pointent vers le moins d'autres nœuds). Il est ainsi possible de représenter de façon linéaire des dépendances entre tâches.

```
>>> list(TopologicalSorter({'A': 'C', 'B': 'C', 'C':
    'D'}).static_order())
['D', 'C', 'A', 'B']
```

Ce tri ne peut bien sûr pas fonctionner si le graphe présente des cycles.

```
>>> list(TopologicalSorter({'A': 'B', 'B': 'A'}).static_order())
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
graphlib.CycleError: ('nodes are in a cycle', ['A', 'B', 'A'])
```

La classe `TopologicalSorter` fournit aussi des méthodes pour construire et trier le graphe itérativement.

5. Types génériques natifs

Depuis que les annotations de types ont été apportées, il est possible de typer les collections suivant le type de leurs éléments à l'aide des types génériques fournis par le module `typing`.

Par exemple `typing.List[int]` identifie une liste de nombres entiers et `typing.Dict[str, int]` un dictionnaire associant des nombres à des chaînes.

```
from typing import List, Dict

def get_values(mapping: Dict[str, int]) -> List[int]:
    return list(mapping.values())

get_values({'foo': 10, 'bar': 42})
```

Cela impliquait donc d'importer ces types génériques depuis le module `typing`. Avec la PEP 585 il est possible de faire cela plus facilement puisque le typage générique devient disponible sur les types natifs.
Ainsi, `typing.List[int]` devient `list[int]` et `typing.Dict[str, int]` devient `dict[str, int]`.

```
def get_values(mapping: dict[str, int]) -> list[int]:
    return list(mapping.values())

get_values({'foo': 10, 'bar': 42})
```

Le module `typing` reste toujours utile puisqu'il fournit des types plus généraux tels que `Iterable` ou `Mapping`.

6. Refonte de l'analyseur syntaxique

Le plus gros changement de Python 3.9 se situe sur l'analyseur syntaxique qui a été complètement refait, comme décrit dans la PEP 617 .

Techniquement il s'agit de passer d'un analyseur (*parser*) *LL(1)* à *PEG* , ce dernier étant plus souple et permettant des constructions plus complexes. Un analyseur syntaxique *LL(1)* fonctionne par une descente récursive (à gauche) ce qui force à catégoriser au plus vite les lexèmes (entre mot-clé et variable par exemple).

Cela ne change rien en pratique, la syntaxe reste la même entre Python 3.8 et Python 3.9, mais ce nouveau type d'analyse prépare des évolutions futures puisqu'il sera possible de créer de nouveaux mot-clés suivant le contexte (sans que cela ne pose problème si le terme est utilisé comme nom de variable dans un autre contexte).
C'est grâce au nouvel analyseur syntaxique que l'on devrait avoir du filtrage par motif (PEP 634) en Python 3.10 par exemple, et d'autres nouveautés suivront.

Vous ne devriez donc constater aucune différence à l'exécution du code, il reste toutefois possible en Python 3.9 d'utiliser le vieil analyseur syntaxique à l'aide de l'option `-X oldparser` fournie à l'interpréteur ou de la variable d'environnement `PYTHONOLDPARSER=1`.

7. Avertissements de dépréciation

> Pensez à vérifier les alertes `DeprecationWarning` qui pourraient se produire dans votre code.

Lors de la transition depuis Python 2, plusieurs fonctions ont été dépréciées mais conservées par rétro-compatibilité.
C'est le cas par exemple des classes de `collections.abc` disponibles directement dans `collections` : utiliser `collections.Mapping` plutôt que `collections.abc.Warning` produit une alerte.

Depuis cette année, Python 2.7 n'est plus supportée et il n'est donc plus utile d'assurer cette rétro-compatibilité. Ainsi, l'alias `collections.Mapping` cité en exemple disparaîtra en Python 3.10, ce qui fera échouer les codes qui continuent à l'utiliser. Prenez donc garde aux alertes de dépréciation et pensez à corriger votre code.

Utilisez le mode développement de Python (`-X dev`) dans votre environnement de test pour remarquer directement ces alertes et faciliter la transition d'une version à une autre.

8. Autres changements

D'autres changements plus mineurs ont été apportés au langage, par exemple :

- PEP 614 —Toute expression est maintenant valide en tant que décorateur.
- `''.replace('', s, n)` renvoie maintenant `s` plutôt qu'une chaîne vide quand `n` n'est pas nul, gagnant en cohérence avec `''.replace('', s)` qui avait déjà ce comportement.
- La variable spéciale `__file__` pointe maintenant vers le chemin absolu du fichier.
- La fonction `__import__` ne lève plus de `ValueError` mais des `ImportError`.
- En mode de développement, les paramètres `encoding` et `errors` des fonctions gérant des conversions d'encodage (`open`, `str.encode`, `bytes.decode`, etc.) sont maintenant vérifiés pour s'assurer qu'ils contiennent des valeurs valides.

L'ensemble des nouveautés apportées par cette version peut être consultée sur la page de documentation dédiée ou dans le *changelog* .

Voilà pour Python 3.9, vous pouvez retrouver plus d'information sur la page dédiée de la documentation .

Cette version est disponible au téléchargement sur le site officiel de Python ou dans votre gestionnaire de paquets favori.

Il faudra attendre la version 3.10 pour de nouvelles fonctionnalités *révolutionnaires* sur le langage, notamment l'arrivée d'un filtrage par motif (*pattern matching*) structurel avec la PEP 634 .

N'hésitez pas à profiter des commentaires pour toute question que vous auriez sur cette version de Python ou ses évolutions futures.