ENTWANNE

Zeste de savoir

Sortie de Python 3.10

12 octobre 2021

Table des matières

Introduction

Le lundi 4 octobre 2021 a marqué l'histoire de l'informatique :

— Facebook et tous les services associés ont subi une panne de plus de 6 heures ;
— Python a sorti sa version 3.10, dont la fonctionnalité phare était attendue/demandée depuis des années.

On ne va pas se mentir, je vous en parlais en conclusion de mon article l'année dernière, la principale évolution apportée par la version 3.10 de Python est le filtrage par motif (ou *pattern-matching*), qui avait régulièrement été demandé mais est revenu sur le devant de la scène lors du changement d'analyseur syntaxique en Python 3.9.

Ce changement ne doit pour autant pas occulter d'autres nouveautés mineures de Python 3.10, évoquées dans les autres sections.

1. Filtrage par motif (pattern matching)

Le filtrage par motif est une construction qui existe dans de nombreux langages, du simple `switch`/`case` en C jusqu'aux plus puissants `match` d'OCaml ou de Rust, ils permettent de décrire des structures conditionnelles où plusieurs valeurs sont testées successivement.

Cette fonctionnalité avait longtemps été demandée en Python, mais toujours refusée au motif qu'il était difficile d'introduire un nouveau mot-clé (`switch`? `case`? `match`?) pour cela, car cela casserait les codes utilisant ce mot comme nom de variable ou de fonction.
Puis vint Python 3.9 et son changement d'analyseur syntaxique ouvrant la voie à ce nouveau mot-clé : il était maintenant possible de ne définir un mot-clé que selon un contexte particulier, et de garder ce mot utilisable comme nom de variable/fonction dans les autres contextes.

1.1. `match` / `case`

C'est ainsi que les mots-clés `match` et `case` ont été choisis pour mettre en œuvre le filtrage par motif en Python.
Un bloc `match`/`case` consiste donc à tester la valeur d'une variable selon certains critères, et à exécuter le code du premier critère correspondant.

```
cmd = input('> ')

match cmd.split():
    case ['help']:
        print('Commands:\n* help\n* hello\n* exit')
    case ['hello']:
        print('Hello World!')
    case ['exit']:
        print('Exiting')
```

```
> hello
Hello World!
```

Mais le filtrage par motif va bien au-delà de simples conditions puisqu'il permet justement de reconnaître… des motifs. Dans le code ci-dessus, les motifs consistent juste à vérifier l'égalité entre notre variable et différentes valeurs, mais on peut par exemple imaginer un motif validant plusieurs valeurs en utilisant l'opérateur d'union `|`.

```
match cmd.split():
    case ['help' | '?']:
        print('Commands:\n* help\n* hello\n* exit')
    case ['hello']:
        print('Hello World!')
```

```
    case ['exit' | 'quit']:
        print('Exiting')
```

```
> ?
Commands:
* help
* hello
* exit
```

1.2. Capture de variables

Mieux encore, les motifs peuvent capturer des variables. Ainsi, on peut ajouter un motif `[other]` qui correspondra à n'importe quelle autre commande, et qui récupérera la commande en question dans une variable `other`.
De même qu'on peut remplacer notre motif `['hello']` par `['hello', name]` pour correspondre aux commandes `hello xxx` et automatiquement récupérer cet argument `xxx` dans une variable `name`.

```
match cmd.split():
    case ['help' | '?']:
        print('Commands:\n* help\n* hello\n* exit')
    case ['hello', name]:
        print(f'Hello {name}!')
    case ['exit' | 'quit']:
        print('Exiting')
    case [other]:
        print(f'Unknown command {other}')
```

```
> hello Clem
Hello Clem!
```

```
> coucou
Unknown command coucou
```

On peut aussi utiliser les syntaxes d'*unpacking* pour récupérer tous les éléments d'une liste avec `[other, *rest]`.

```
match cmd.split():
    case ['help' | '?']:
        print('Commands:\n* help\n* hello\n* exit')
    case ['hello', name]:
        print(f'Hello {name}!')
    case ['exit' | 'quit']:
        print('Exiting')
    case [other, *rest]:
        print(f'Unknown command {other} with args {rest}')
```

```
> ls -l -a
Unknown command ls with args ['-l', '-a']
```

1.3. *Wildcard*

Le nom `_` peut être utilisé pour réaliser un motif sans capturer de nom. Ce motif correspond à toute valeur possible, on l'appelle alors un *wildcard* (ou joker).

Ainsi `[_]` validera une liste d'un seul élément, sans affecter cet élément à une variable.

```
def test_list(values):
    match values:
        case []:
            print('La liste est vide')
        case [_]:
            print('La liste contient un élément')
        case _:
            print('La liste contient plusieurs éléments')
```

```
>>> test_list([])
La liste est vide
>>> test_list([1])
La liste contient un élément
>>> test_list([1, 2])
La liste contient plusieurs éléments
```

1.4. Déstructuration

Le filtrage par motif sert aussi à déstructurer des objets complexes vers des variables simples, de la même manière que le fait l'*unpacking* pour une liste.

1. Filtrage par motif (pattern matching)

Imaginons une classe `Point` composée de deux champs `x` et `y` :

```
from dataclasses import dataclass

@dataclass
class Point:
    x: int
    y: int
```

Il est possible, avec un motif, de reconnaître un tel objet `Point` et d'en extraire les attributs `x` et `y`.

```
def print_point(p):
    match p:
        case Point(x, y):
            print(f"Abscisse: {x}, ordonnée: {y}")
```

```
>>> print_point(Point(3, 5))
Abscisse: 3, ordonnée: 5
```

Et l'on peut encore préciser des valeurs particulières pour reconnaître certains cas spécifiques.

```
def print_point(p):
    match p:
        case Point(0, 0):
            print("Origine du repère")
        case Point(x, 0):
            print(f"Point sur l'axe des abscisses, abscisse: {x}")
        case Point(0, y):
            print(f"Point sur l'axe des ordonnées, ordonnée: {y}")
        case Point(x, y):
            print(f
                "Point quelconque, abscisse: {x}, ordonnée: {y}")
```

```
>>> print_point(Point(0, 0))
Origine du repère
>>> print_point(Point(0, 3))
Point sur l'axe des ordonnées, ordonnée: 3
>>> print_point(Point(5, 0))
Point sur l'axe des abscisses, abscisse: 5
>>> print_point(Point(5, 3))
```

```
Point quelconque, abscisse: 5, ordonnée: 3
```

Pour plus d'informations au sujet du filtrage par motif, je vous invite à consulter ce tutoriel de la PEP 636 , dont sont inspirés les exemples de cet article.

2. Autres nouveautés

Faisons maintenant un tour d'horizon des autres nouveautés apportées par Python 3.10.

2.1. Vérification optionnelle de la taille pour `zip` (PEP 618)

Vous connaissez la fonction `zip` ? C'est une fonction qui permet d'assembler plusieurs itérables pour les parcourir simultanément.

```
>>> words = ['abc', 'def', 'ghi']
>>> numbers = [4, 5, 5]
>>> for word, number in zip(words, numbers):
...     print(number, '-', word)
...
4 - abc
5 - def
5 - ghi
```

Que fait cette fonction si nos itérables n'ont pas tous la même taille (disons que `words` contienne 4 éléments alors que `numbers` n'en contient que 3) ? Elle s'arrête au premier terminé, silencieusement.

```
>>> words.append('jkl')
>>> for word, number in zip(words, numbers):
...     print(number, '-', word)
...
4 - abc
5 - def
5 - ghi
```

Depuis Python 3.10, `zip` vient avec un paramètre optionnel booléen `strict` qui permet de lever une erreur dans un tel cas pour avertir que tous les itérables n'ont pas la même taille.

```
>>> for word, number in zip(words, numbers, strict=True):
...     print(number, '-', word)
...
4 - abc
5 - def
5 - ghi
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: zip() argument 2 is shorter than argument 1
```

Cette erreur ne survient qu'en fin d'itération car les itérables peuvent être de taille indéterminée, voire infinie.

Notez que pour avoir le comportement inverse (itérer jusqu'au plus long), il existe aussi la fonction `zip_longest` du module `itertools` .

2.2. Parenthèses des gestionnaires de contexte

Il s'agit plus d'une correction de bug que d'une réelle fonctionnalité, mais c'est un changement suffisamment pratique pour l'évoquer ici.

Quand vous utilisez des gestionnaires de contexte , il arrive que vous souhaitiez en ouvrir plusieurs en même temps.

```
with open('input', 'r') as finput, open('output', 'w') as foutput:
    ...
```

Mais dans une volonté d'aérer un peu le code, vous pourriez vouloir placer les deux gestionnaires sur des lignes distinctes. Jusqu'à Python 3.9, il fallait utiliser pour cela un caractère antislash (\) en fin de ligne, ce qui pouvait avoir pour effet de casser l'indentation de votre éditeur.

```
with open('input', 'r') as finput, \
     open('output', 'w') as foutput:
    ...
```

Il est maintenant possible de placer l'ensemble des contextes à ouvrir dans des parenthèses et d'éviter les soucis d'alignement.

```
with (
        open('input', 'r') as finput,
        open('output', 'w') as foutput,
):
    ...
```

Cette évolution était en réalité déjà présente en Python 3.9, si vous utilisiez le nouvel analyseur syntaxique (par défaut).
Python 3.10 entérine ce nouvel analyseur et donc cette syntaxe.

2.3. Fonctions `aiter` et `anext`

Il existe en Python des fonctions `iter` et `next`, respectivement pour obtenir un itérateur sur un itérable, et pour avancer un itérateur.

```
>>> values = range(10)
>>> it = iter(values)
>>> next(it)
0
>>> next(it)
1
>>> next(it)
2
```

Il n'existait jusqu'alors pas de telles fonctions pour les itérables asynchrones, c'est maintenant corrigé avec l'arrivée des fonctions `aiter` et `anext`.

```
async def get_values():  # get_values est ici un générateur asynchrone
    yield 1
    yield 2
    yield 3

async def main():
    it = aiter(get_values())
    print(await anext(it))
    print(await anext(it))
```

Dans la même veine, une fonction `aclosing` est ajoutée au module `contextlib`, similaire à `closing` mais pour des objets asynchrones : elle renvoie donc un gestionnaire de contexte asynchrone appelant automatiquement la coroutine `aclose` de l'objet à la fin du bloc.

2.4. *dataclasses* et arguments *keyword-only*

Les *dataclasses* (classes de données) sont une nouveauté de Python 3.7, elles permettent d'écrire facilement des classes dédiées à simplement contenir des données, sans avoir à gérer manuellement les attributs.

```
import dataclasses

@dataclasses.dataclass
class Point:
    x: int
    y: int

p = Point(1, 2)
print(p.x, p.y)
```

On le voit, `Point` est appelée ici avec des arguments positionnels, mais les arguments nommés sont aussi autorisés (`Point(x=1, y=2)`).

Avec Python 3.10 il devient possible d'obliger l'utilisation des arguments nommés pour certains ou tous les attributs. On peut ainsi utiliser `kw_only=True` lors de la définition de la *dataclass* pour l'appliquer à tous les attributs.

```
@dataclasses.dataclass(kw_only=True)
class Point:
    x: int
    y: int
```

Si on veut faire du cas par cas, on peut utiliser le champ `field` du module `dataclasses` pour préciser la valeur de `kw_only` pour les champs affectés.

```
@dataclasses.dataclass
class Point:
    x: int
    y: int = dataclasses.field(kw_only=True)
```

Enfin, il est aussi possible d'utiliser l'annotation `KW_ONLY` du même module sur un attribut spécial `_` pour signifier que tous les champs qui suivent sont à arguments nommés seulement.

```
@dataclasses.dataclass
class Point:
    x: int
    _: dataclasses.KW_ONLY
```

```
    y: int
```

2.5. Amélioration des messages d'erreurs

Les messages d'erreurs de Python ont été grandement améliorés en 3.10 : l'interpréteur nous donne maintenant plus d'informations pour essayer d'identifier nos erreurs.

```
>>> foo = 10
>>> print(fooo)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'fooo' is not defined. Did you mean: 'foo'?
>>> (1, 2, 3 4)
  File "<stdin>", line 1
    (1, 2, 3 4)
             ^^^
SyntaxError: invalid syntax. Perhaps you forgot a comma?
```

De plus, la PEP 626 améliore la précision des numéros de lignes renvoyés dans les erreurs.

2.6. En vrac

— Le type `int` comporte maintenant une méthode `bit_count` pour compter le nombre de bits à 1 du nombre entier.

```
>>> (1).bit_count()
1
>>> (2).bit_count()
1
>>> (3).bit_count()
2
>>> (7).bit_count()
3
```

— Le module `distutils` est maintenant déprécié (au profit de `setuptools` et `packaging`).
— La syntaxe permettant de ne pas mettre d'espaces entre nombres et mots-clés (`5in range(10)`) est maintenant dépréciée.

3. Nouveautés sur le typage (type hints)

Les annotations de types sont une syntaxe optionnelle de Python et permettent de préciser les types des variables et paramètres. Cela sert à des outils tels que *mypy* pour faire de l'analyse

statique sur le code (vérifier que toutes les opérations sont faites en concordance avec les types des données).

Ces annotations sont en constante évolution et chaque version de Python y apporte donc son lot de nouveautés.

3.1. Simplification des unions (PEP 604)

Pour indiquer qu'une variable pouvait être de plusieurs types différents (par exemple un nombre entier ou un flottant), il fallait auparavant utiliser l'annotation `Union[int, float]` (du module `typing`).

```
1 from typing import Union
2
3 def invert(x: Union[int, float]) -> float:
4     return 1 / x
5
6 print(invert(1))
7 print(invert(2))
```

Il est maintenant possible de simplement écrire cette union comme `int | float`.

```
1 def invert(x: int | float) -> float:
2     return 1 / x
```

`int | float` est un objet `UnionType`, et on remarquera notamment qu'il est utilisable pour des appels à `isinstance` afin de tester le type d'une valeur.

```
1 >>> int | float
2 int | float
3 >>> isinstance(42, int | float)
4 True
5 >>> isinstance(3.5, int | float)
6 True
7 >>> isinstance('abc', int | float)
8 False
```

Plus d'informations sont à trouver dans la PEP 604 .

3.2. Alias explicites (PEP 613)

Python 3.10 propose une nouvelle syntaxe pour les alias de types. Un alias est une variable qui permet de référencer une annotation de type, afin de la réutiliser à d'autres endroits. Pour reprendre l'exemple précédent, on pourrait avoir un alias `Real` pour le type `int | float`.

```
1 Real = int | float
2
3 def invert(x: Real) -> float:
4     return 1 / x
```

Le soucis dans ce code c'est qu'on ne sait pas bien à la lecture de `Real` quelle est son intention et à quoi il va servir. Les alias explicites résolvent ce problème, on peut maintenant annoter `Real` avec `TypeAlias` (du module `typing`) pour indiquer explicitement qu'il s'agit d'un alias et qu'il sera utilisé pour des annotations.

```
1 from typing import TypeAlias
2
3 Real: TypeAlias = int | float
```

Voir la PEP 613 pour en apprendre plus sur les alias explicites.

3.3. Annotations des *callables* (PEP 612)

Pour terminer sur les annotations de types, je vais vous parler des *callables* (objets appelables, les fonctions par exemple).

Il est souvent difficile de bien typer un paramètre qui peut recevoir une fonction : à quel niveau de détail faut-il aller ? Il existe pour cela le type `Callable` du module `collections.abc` qui permet d'annoter un *callable* avec les types de ses paramètres et son type de retour (par exemple `Callable[[int, int], int]` pour une fonction d'addition entre deux entiers).

Mais comment typer une fonction recevant un *callable* quelconque en paramètre si celle-ci doit renvoyer un *callable* du même type (comme c'est souvent le cas des décorateurs) ?

La PEP 612 répond à cela en ajoutant un utilitaire `ParamSpec` pour définir une spécification réutilisable pour la liste des paramètres d'une fonction.

```
1 from collections.abc import Callable
2 from typing import ParamSpec
3
4 P = ParamSpec('P')
5
6 def decorator(f: Callable[P, None]) -> Callable[P, None]:
```

```
    def inner(*args: P.args, **kwargs: P.kwargs) -> None:
        print('before')
        f(*args, **kwargs)
        print('after')
    return inner
```

La fonction `decorator` prend donc en argument un *callable* quelconque (qui renvoie `None`) et renvoie un `callable` du même type. Ainsi, `decorator` appelée sur une fonction `Callable[[int], None]` renverra une fonction `Callable[[int], None]`.

La PEP apporte aussi l'annotation `Concatenate` pour appliquer des transformations à une liste de paramètres (ajouter un paramètre à la liste par exemple)

Conclusion

Vous trouverez plus d'informations au sujet de la version 3.10 de Python sur cette page de documentation .

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

La prochaine version (Python 3.11) est déjà en cours de développement, et sa sortie est prévue pour le 3 octobre 2022 .

Pour toute question au sujet de Python 3.10 ou du filtrage par motif, n'hésitez pas à utiliser l'espace de commentaires sous cet article, ou le forum.

L'image de l'article est tirée de la page release Python 3.10.0 .