ENTWANNE

Zeste de savoir

Sortie de Python 3.11

27 octobre 2022

Table des matières

Introduction

Le rythme de publication des nouvelles versions de Python étant maintenant annuel, après Python 3.10 l'année dernière c'est donc ce lundi 24 octobre 2022 qu'est sortie la version 3.11 de Python.

Cette version apporte quelques nouveautés mais est surtout attendue pour le gain en performances de l'interpréteur. Voyons tout de suite plus en détails le contenu de cette version.

1. Performances de CPython

Cette version était particulièrement attendue pour la promesse du gain en rapidité de l'interpréteur. Elle marque en effet une étape importante d'un travail de fond pour améliorer les performances de CPython (l'implémentation standard de l'interpréteur Python), elle est annoncée comme en moyenne 1,25 fois plus rapide que CPython 3.10 (un gain de 10 à 60% suivant les cas).

Ces gains sont dus à un ensemble d'optimisations dont voici les principales :

— Le démarrage de l'interpréteur est plus rapide grâce à une allocation statique pour importer les modules Python essentiels.
— L'exécution est plus rapide elle aussi grâce à l'optimisation des *frames* (les cadres qui définissent le contexte courant lors des appels de fonctions) qui sont maintenant réduites au strict nécessaire et réutilisées pour éviter de trop nombreuses allocations.

— On note que les anciens styles de *frames* sont toujours disponibles à la demande si nécessaires pour des outils de débogage ou d'inspection par exemple, mais aucune *frame* ne sera plus jamais créée pour la plupart des codes.
— Le coût des blocs `try` pour gérer les exceptions est maintenant pratiquement nul quand aucune exception n'est levée.
— Les appels de fonctions sont aussi optimisés (*inlined*) pour ne pas consommer d'espace mémoire au niveau de la pile (*stack*) du processus, c'est-à-dire que le coût d'un appel de fonction est considérablement réduit.
— Des chemins d'exécution rapide sont générés à la volée pour spécialiser les types (et bénéficier d'instructions optimisées pour ces types) quand un code est fréquemment utilisé avec les mêmes types de données, voir la PEP 659 à ce sujet.

On note qu'il n'existe pour le moment aucune compilation JIT pour optimiser les instructions en temps réel.

2. Gestion des exceptions

2.1. Groupes d'exception

En dehors de ces optimisations qui ne représentent pas une nouveauté en tant que tel (Python continue de fonctionner à peu près de la même manière, il est juste plus rapide), la principale avancée se retrouve sur la gestion des exceptions multiples.
Avec la PEP 654 , Python apporte en effet une nouvelle manière de gérer ces exceptions.

Il est courant de rencontrer des situations où plusieurs erreurs distinctes surviennent et qu'on aimerait toutes voir remonter (comme dans des cas de validation—erreurs sur plusieurs champs, ou de gestions de tâches—erreurs dans différentes tâches), sans devoir s'arrêter à la première erreur ou en ignorer certaines.
La manière de gérer cela est de construire un nouveau type d'exception servant de conteneur à exceptions.

C'est ainsi qu'est introduit `ExceptionGroup` dans la PEP 654, une nouvelle exception qui permet de regrouper plusieurs erreurs.
On construit un groupe en lui précisant un message (chaîne de caractères) et une séquence d'exceptions.

```
>>> exc_group = ExceptionGroup('multiple errors', [
...     ValueError('Incompatible value'),
...     TypeError('Incompatible type'),
... ])
>>> print(exc_group)
multiple errors (2 sub-exceptions)
>>> raise exc_group
  + Exception Group Traceback (most recent call last):
  |   File "<stdin>", line 1, in <module>
  | ExceptionGroup: multiple errors (2 sub-exceptions)
  +-+---------------- 1 ----------------
```

```
    | ValueError: Incompatible value
    +---------------- 2 ----------------
    | TypeError: Incompatible type
    +------------------------------------
```

Mieux encore, ces groupes peuvent être imbriqués pour représenter une arborescence d'erreurs.

```
>>> big_group = ExceptionGroup('critical', [exc_group,
    OSError('No processor found')])
>>> print(big_group)
critical (2 sub-exceptions)
>>> raise big_group
  + Exception Group Traceback (most recent call last):
  |   File "<stdin>", line 1, in <module>
  | ExceptionGroup: critical (2 sub-exceptions)
  +-+---------------- 1 ----------------
    | Exception Group Traceback (most recent call last):
    |   File "<stdin>", line 1, in <module>
    | ExceptionGroup: multiple errors (2 sub-exceptions)
    +-+---------------- 1 ----------------
      | ValueError: Incompatible value
      +---------------- 2 ----------------
      | TypeError: Incompatible type
      +------------------------------------
    +---------------- 2 ----------------
    | OSError: No processor found
    +------------------------------------
```

Mais ce nouveau type en lui-même ne suffit pas à régler le problème posé dans cette PEP. En effet nous disposons ici d'un groupe d'exceptions que l'on peut lever (avec `raise exc_group`), mais comment faire ensuite pour traiter les différentes exceptions qu'il contient ?

Une solution serait d'utiliser un `except ExceptionGroup as eg:`, de regarder si le groupe `eg` contient l'exception que l'on souhaite traiter ici (par exemple `ValueError`) et de laisser remonter les autres exceptions.
Ce qui serait plutôt fastidieux et propice aux erreurs :

- Il faudrait forcément un traitement particulier et systématique pour le type `ExceptionGroup`.
- Il faudrait dupliquer le traitement des erreurs si l'on veut pouvoir traiter à la fois une `ValueError` et un groupe contenant une `ValueError`.
- On pourrait trop facilement oublier de faire remonter les erreurs restantes d'un groupe et donc les faire passer involontairement sous silence.

C'est pourquoi Python 3.11 introduit aussi un nouveau mot-clé, `except*`, afin de régler les problèmes évoqués.
Un `except* ValueError:` permet en effet d'attraper toutes les exceptions `ValueError`, qu'elles

soient levées seules ou issues d'un groupe, et dans le cas d'un groupe de laisser remonter le reste des exceptions.

```
>>> try:
...     raise exc_group
... except* ValueError:
...     print('Ignoring ValueError')
...
Ignoring ValueError
  + Exception Group Traceback (most recent call last):
  |   File "<stdin>", line 2, in <module>
  |   File "<stdin>", line 1, in <module>
  | ExceptionGroup: multiple errors (1 sub-exception)
  +-+---------------- 1 ----------------
    | TypeError: Incompatible type
    +------------------------------------
```

On constate bien que la `ValueError` de notre groupe est traitée et que la `TypeError` continue de survenir. Il est ainsi possible de placer plusieurs `except*` à la suite pour traiter les différentes exceptions possibles (même au sein d'un même groupe).

```
>>> try:
...     raise big_group
... except* ValueError:
...     print('Ignoring ValueError')
... except* TypeError:
...     print('Ignoring TypeError')
... except* OSError:
...     print('Ignoring OSError')
...
Ignoring ValueError
Ignoring TypeError
Ignoring OSError
```

On peut aussi placer un `as` derrière `except*` (comme on le ferait avec `except`) pour récupérer l'instance de l'exception.
Cependant, comme `except*` capture potentiellement plusieurs exceptions d'un même type (toutes les `ValueError` du groupe par exemple) il n'est pas possible de pointer vers une instance particulière de ces exceptions. Ainsi l'objet renvoyé est en fait une instance d'`ExceptionGroup` contenant uniquement les exceptions correspondantes.

```
>>> try:
...     raise ExceptionGroup('errors', [ValueError(1),
    ValueError(2)])
... except* ValueError as eg:
```

```
...     print(f'caught {eg!r}')
...
caught ExceptionGroup('errors', [ValueError(1), ValueError(2)])
```

```
>>> try:
...     raise ExceptionGroup('errors', [ValueError(1),
    TypeError('foo')])
... except* ValueError as eg:
...     print(f'caught {eg!r}')
... except* TypeError as eg:
...     print(f'caught {eg!r}')
...
caught ExceptionGroup('errors', [ValueError(1)])
caught ExceptionGroup('errors', [TypeError('foo')])
```

Et ce même s'il s'agit au départ d'une exception simple.

```
>>> try:
...     raise ValueError
... except* ValueError as eg:
...     print(f'caught {eg!r}')
...
caught ExceptionGroup('', (ValueError(),))
```

Si nous nous intéressons de plus près aux objets `ExceptionGroup` nous pouvons voir qu'ils possèdent un attribut `exceptions` (la séquence des exceptions inclues dans le groupe).

```
>>> exc_group.exceptions
(ValueError('Incompatible value'), TypeError('Incompatible type'))
>>> big_group.exceptions
(
    ExceptionGroup(
        'multiple errors',
        [ValueError('Incompatible value'), TypeError('Incompatible
    type')],
    ),
    OSError('No processor found'),
)
```

Ces objets possèdent aussi deux méthodes intéressantes, `subgroup` et `split`.
`subgroup` est appelée avec une condition (un prédicat) en argument et permet de filtrer les exceptions du groupe pour ne renvoyer que celles qui remplissent la condition (un sous-groupe

formé de ces exceptions). La méthode gère très bien l'arborescence des exceptions dans le groupe.

```
>>> exc_group.subgroup(lambda e: True)
ExceptionGroup(
    'multiple errors',
    [ValueError('Incompatible value'), TypeError('Incompatible
    type')],
)
>>> exc_group.subgroup(lambda e: isinstance(e, ValueError))
ExceptionGroup(
    'multiple errors',
    [ValueError('Incompatible value')],
)
>>> big_group.subgroup(lambda e: isinstance(e, TypeError))
ExceptionGroup(
    'critical',
    [ExceptionGroup('multiple errors', [TypeError('Incompatible
    type')])],
)
```

Quand aucune exception ne correspond dans le groupe, la méthode renvoie simplement `None`.

```
>>> exc_group.subgroup(lambda e: False)
```

La méthode `split` est similaire mais renvoie un tuple de 2 éléments : le groupe des exceptions qui correspondent à la condition et le groupe des exceptions exclues. Chaque groupe pouvant valoir `None` s'il est vide.

```
>>> exc_group.split(lambda e: isinstance(e, ValueError))
(
    ExceptionGroup('multiple errors', [ValueError('Incompatible
    value')]),
    ExceptionGroup('multiple errors', [TypeError('Incompatible
    type')]),
)
>>> exc_group.split(lambda e: isinstance(e, Exception))
(
    ExceptionGroup(
        'multiple errors',
        [ValueError('Incompatible value'), TypeError('Incompatible
    type')],
    ),
    None,
)
```

Et comme il est relativement courant de filtrer les exceptions par type, la condition donnée à `subgroup`/`split` peut simplement être un type ou un tuple de types.

```
>>> exc_group.subgroup(TypeError)
ExceptionGroup('multiple errors', [TypeError('Incompatible type')])
>>> big_group.split((ValueError, OSError))
(
    ExceptionGroup(
        'critical',
        [
            ExceptionGroup('multiple errors',
    [ValueError('Incompatible value')]),
            OSError('No processor found'),
        ]
    ),
    ExceptionGroup(
        'critical',
        [ExceptionGroup('multiple errors',
    [TypeError('Incompatible type')])],
    ),
)
```

C'est d'ailleurs cette méthode `split` qui est utilisée en interne pour la machinerie d'`except*` qui pourrait explicitement être réalisée comme suit avec un simple `except`.

```
>>> try:
...     raise exc_group
... except ExceptionGroup as eg:
...     match, subgroup = eg.split(ValueError)
...     if match is not None:
...         print('Ignoring ValueError')
...     if subgroup is not None:
...         raise subgroup
...
Ignoring ValueError
  + Exception Group Traceback (most recent call last):
  |   File "<stdin>", line 8, in <module>
  |   File "<stdin>", line 2, in <module>
  |   File "<stdin>", line 2, in <module>
  |   File "<stdin>", line 1, in <module>
  | ExceptionGroup: multiple errors (1 sub-exception)
  +-+---------------- 1 ----------------
    | TypeError: Incompatible type
    +------------------------------------
```

Si vous prévoyez d'utiliser les groupes d'exceptions dans une bibliothèque que vous développez, prenez garde au fait que cela cassera la compatibilité pour vos utilisateurs qui devront maintenant traiter correctement ces groupes avec `except*`.

La PEP 654 fourmille d'exemples en tout genre, je vous conseille d'aller y jeter un œil pour mieux comprendre cette nouveauté.

2.2. Gestion plus fine des erreurs

Python 3.11 apporte d'autres changements dans la gestion des exceptions.

On remarque notamment que les erreurs remontées sont maintenant plus claires en pointant directement l'endroit précis (ligne et colonne) dans le fichier qui a causé l'erreur grâce à la PEP 657 .

```
Traceback (most recent call last):
  File "test.py", line 1, in <module>
    print(3 + 5 * None)
              ~~^~~~~~
TypeError: unsupported operand type(s) for *: 'int' and 'NoneType'
```

2.3. Notes de contexte

Enfin la PEP 678 ajoute une méthode `add_note` aux exceptions pour les enrichir avec des notes de contexte.

```
>>> def division(a, b):
...     try:
...         return a / b
...     except ZeroDivisionError as exc:
...         exc.add_note(f'When computing division({a!r}, {b!r})')
...         raise exc
...
>>> division(10, 0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 6, in division
  File "<stdin>", line 3, in division
ZeroDivisionError: division by zero
When computing division(10, 0)
```

3. Introduction de la tomllib

Avant la version 3.11 il était déjà possible de gérer le format de fichiers TOML en Python, à l'aide d'un paquet externe.

Cela posait problème puisque le langage était devenu celui par défaut pour la configuration, notamment en ce qui concerne la construction de paquets. Il a donc été décidé d'intégrer à la bibliothèque standard une implémentation minimale du langage afin que les outils de *packaging* puissent en bénéficier sans dépendance externe. C'est la PEP 680 qui introduit ce changement.

Ce module propose des outils dédiés à la lecture de fichiers TOML (et uniquement à la lecture) via ses fonctions `load` et `loads`. C'est l'interface standard des modules de *parsing* en Python.

— La fonction `load` prend un fichier en argument et renvoie un objet représentant les données lues dans ce fichier.
Par exemple avec le fichier suivant :

```
>>> def division(a, b):
...     try:
...         return a / b
...     except ZeroDivisionError as exc:
...         exc.add_note(f'When computing division({a!r}, {b!r})')
...         raise exc
...
>>> division(10, 0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 6, in division
  File "<stdin>", line 3, in division
ZeroDivisionError: division by zero
When computing division(10, 0)
```

Listing 1 – `testfile.toml`

On peut utiliser la fonction `tomllib.load` comme suit.

```
>>> import tomllib
>>> with open('testfile.toml', 'rb') as f:
...     config = tomllib.load(f)
...
>>> config
{'section': {'foo': 'bar', 'x': 42}, 'section2': {'foo':
    {'baz': []}}}
```

On note qu'il faut ouvrir le fichier en mode binaire (`'b'`) pour le passer à la fonction `load`. Cela est dû au fait que l'encodage du texte est géré en interne par le module.

— La fonction `loads` procède de la même manière avec une chaîne de caractères en entrée.

```
>>> tomllib.loads('[x]\ny.z = 0')
{'x': {'y': {'z': 0}}}
```

On note que ces fonctions acceptent aussi un argument `parse_float` qui à la manière du module `json` permet de spécifier le type à utiliser pour gérer les nombres décimaux lus dans le fichier TOML. Il s'agit par défaut du type `float`, le plus simple pour gérer ces nombres mais qui introduit des imprécisions lors des calculs.

```
>>> tomllib.loads('value = 4.2')
{'value': 4.2}
>>> from decimal import Decimal
>>> tomllib.loads('value = 4.2', parse_float=Decimal)
{'value': Decimal('4.2')}
```

Autrement, voici le tableau de conversion des types TOML vers Python :

TOML	Python
string	`str`
integer	`int`
float	`float` (configurable avec `parse_float`)
boolean	`bool`
array	`list`
table	`dict`
offset date-time	`datetime.datetime` avisé (avec `tzinfo`)
local date-time	`datetime.datetime` naïf (`tzinfo=None`)
local date	`datetime.date`
local time	`datetime.time`

Plus d'informations sont à retrouver sur la documentation du module `tomllib` .

4. Du nouveau concernant asyncio

Est-ce que vous connaissez la fonction `gather` du module `asyncio` ?
C'est une fonction qui permet de lancer et d'attendre plusieurs tâches simultanées.

```
>>> import asyncio
>>>
>>> async def coro(name, sleep_duration=1):
...     print(f'enter {name}')
...     await asyncio.sleep(sleep_duration)
...     print(f'exit {name}')
...
>>> async def main():
...     await asyncio.gather(
...         coro('foo', sleep_duration=2),
...         coro('bar'),
...     )
...
>>> asyncio.run(main())
enter foo
enter bar
exit bar
exit foo
```

Il est maintenant possible (et préférable) d'utiliser un `TaskGroup` pour cela. Les groupes s'utilisent comme des gestionnaires de contexte asynchrones (bloc `async with`) et possèdent une méthode `create_task` pour programmer une tâche dans le groupe courant.

```
>>> async def main():
...     async with asyncio.TaskGroup() as group:
...         group.create_task(coro('foo', sleep_duration=2))
...         group.create_task(coro('bar'))
...
>>> asyncio.run(main())
enter foo
enter bar
exit bar
exit foo
```

Le module `asyncio` se dote aussi d'une nouvelle primitive de synchronisation, `Barrier` . Celle-ci permet d'attendre jusqu'à ce qu'un certain nombre de tâches soit retenues par la barrière.

```
>>> async def barrier_coro(barrier, sleep_time):
...     await asyncio.sleep(sleep_time)
...     print('Waiting for barrier')
...     await barrier.wait()
...     print('Released')
...
>>> async def main():
...     barrier = asyncio.Barrier(4)  # Attend 4 tâches
...     async with asyncio.TaskGroup() as group:
...         for i in range(4):
...             group.create_task(barrier_coro(barrier,
    sleep_time=i+1))
...
>>> asyncio.run(main())
Waiting for barrier
Waiting for barrier
Waiting for barrier
Waiting for barrier
Released
Released
Released
Released
```

On notera enfin la nouvelle classe `Runner` qui expose le comportement de la fonction `asyncio.run` (récupérer la boucle événementielle, lancer une tâche et s'arrêter).

```
>>> with asyncio.Runner() as runner:
...     runner.run(main())
...
enter foo
enter bar
exit bar
exit foo
```

5. Ça bouge aussi côté enums

Cette version apporte quelques changements au module `enum`, notamment dans la représentation des membres des énumérations.
En effet, les types `IntEnum` et `IntFlag` héritent maintenant d'une nouvelle classe `ReprEnum` qui définit plus finement comment les membres doivent être représentés ou convertis.

```
>>> import enum
>>>
>>> class LogLevel(enum.IntEnum):
...     DEBUG = 0
...     INFO = 1
...     WARNING = 2
...     ERROR = 3
...
>>> LogLevel.WARNING
<LogLevel.WARNING: 2>
>>> str(LogLevel.WARNING)
'2'
>>> print(LogLevel.WARNING)
2
```

En Python 3.10 le code précédent aurait eu le comportement suivant :

```
>>> LogLevel.WARNING
<LogLevel.WARNING: 2>
>>> str(LogLevel.WARNING)
'LogLevel.WARNING'
>>> print(LogLevel.WARNING)
LogLevel.WARNING
```

On note au passage que le type des énumérations est maintenant `EnumType` et non plus `EnumMeta`, ce dernier étant toutefois conservé pour la rétrocompatibilité.

Est aussi introduit le type `StrEnum` pour représenter des énumérations dont les membres sont des chaînes de caractères et peuvent alors être utilisés comme telles.

```
>>> class ConfigDir(enum.StrEnum):
...     LOCAL = '.'
...     USER = '~/.foobar'
...     SYSTEM = '/etc/foobar'
...
>>> ConfigDir.USER
<ConfigDir.USER: '~/.foobar'>
>>> print(ConfigDir.USER)
~/.foobar
>>> ConfigDir.USER + '/config.toml'
'~/.foobar/config.toml'
```

Une nouvelle énumération `FlagBoundary` (`STRICT` / `CONFORM` / `EJECT` / `KEEP`) est créée afin de pouvoir gérer plus finement les cas de valeurs au-delà des limites pour les énumérations de type *flag*.

```
>>> class Options(enum.Flag, boundary=enum.FlagBoundary.STRICT):
...     VERBOSE = enum.auto()
...     SORTED = enum.auto()
...     COLORED = enum.auto()
...
>>> Options(0b1)
<Options.VERBOSE: 1>
>>> Options(0b111)
<Options.VERBOSE|SORTED|COLORED: 7>
>>> Options(0b1111)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/lib/python3.11/enum.py", line 695, in __call__
    return cls.__new__(cls, value)
           ^^^^^^^^^^^^^^^^^^^^^^^
  File "/usr/lib/python3.11/enum.py", line 1119, in __new__
    raise exc
  File "/usr/lib/python3.11/enum.py", line 1096, in __new__
    result = cls._missing_(value)
             ^^^^^^^^^^^^^^^^^^^^
  File "/usr/lib/python3.11/enum.py", line 1381, in _missing_
    raise ValueError(
ValueError: <flag 'Options'> invalid value 15
    given 0b0 1111
  allowed 0b0 0111
```

Enfin on remarque que plusieurs décorateurs sont ajoutés au module afin de mieux contrôler les valeurs possibles :

— `verify` permet d'appliquer des contraintes définies dans `EnumCheck` (`UNIQUE` / `CONTINOUS` / `NAMED_FLAGS`) sur les membres de l'énumération.

```
>>> @enum.verify(enum.EnumCheck.CONTINUOUS)
... class LogLevel(enum.IntEnum):
...     DEBUG = 0
...     INFO = 1
...     ERROR = 3
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/lib/python3.11/enum.py", line 1828, in __call__
    raise ValueError(('invalid %s %r: missing values %s' % (
ValueError: invalid enum 'LogLevel': missing values 2
```

— Les fonctions `member` et `nonmember` permettent de définir explicitement ce qui doit être considéré comme membre de l'énumération et ce qui ne doit pas l'être.

```
>>> class MyEnum(enum.Enum):
...     X = 0  # default = member
...     Y = enum.member(1)
...     Z = enum.nonmember(2)
...
...     # default = nonmember
...     def foo(self):
...         return 'foo'
...
...     @enum.member
...     def bar(self):
...         return 'bar'
...
...     @enum.nonmember
...     def baz(self):
...         return 'baz'
...
>>> MyEnum.X
<MyEnum.X: 0>
>>> MyEnum.Y
<MyEnum.Y: 1>
>>> MyEnum.Z
2
>>> MyEnum.foo
<function MyEnum.foo at 0x7f8d9de802c0>
>>> MyEnum.bar
<MyEnum.bar: <function MyEnum.bar at 0x7f8d9de80900>>
>>> MyEnum.bar.value
<function MyEnum.bar at 0x7f8d9de80900>
>>> MyEnum.baz
<function MyEnum.baz at 0x7f8d9de800e0>
```

— `property` est conçu pour fonctionner de la même manière que le décorateur `property` habituel mais plus adapté aux énumérations, qui permet par exemple de gérer les conflits entre propriétés et membres de l'énumération.

```
>>> class BaseEnum(enum.Enum):
...     @enum.property
...     def test(self):
...         return self.value == 'test'
...
>>> class EnvEnum(BaseEnum):
...     prod = 'prod'
...     test = 'test'
...
>>> EnvEnum.prod
<EnvEnum.prod: 'prod'>
>>> EnvEnum.test
```

```
<EnvEnum.test: 'test'>
>>> EnvEnum.prod.test
False
>>> EnvEnum.test.test
True
```

— `global_enum` permet de décorer une énumération pour que ses membres apparaissent comme membres du module plutôt que de l'énumération.

```
>>> @enum.global_enum
... class LogLevel(enum.IntEnum):
...     DEBUG = 0
...     INFO = 1
...     WARNING = 2
...     ERROR = 3
...
>>> LogLevel.INFO
__main__.INFO
```

6. Nouveautés sur le typing

Chaque version apporte son lot de nouveautés concernant le *typing*. Voyons alors ce qui est ajouté en Python 3.11.

La PEP 673 ajoute le nouveau type `Self` qui permet d'annoter les méthodes d'une classe afin de préciser plus facilement qu'elles attendent/renvoient un objet de cette classe. Cette annotation est bien sûr facultative pour le paramètre `self` qui est l'instance courante de la classe.

```
class User:
    ...

    @classmethod
    def load(cls, external_id) -> Self:
        data = DB.get_one('users', external_id=external_id)
        return cls(**data)

    def link(self, other: Self):
        DB.create('user_links', user1=self.id, user2=other.id)
```

Avec la PEP 655 c'est aussi le type `TypedDict` qui est amélioré afin de pouvoir préciser plus facilement les champs requis (`Required`) ou non (`NotRequired`) dans un dictionnaire.

```python
from typing import TypedDict

class User(TypedDict):
    email: Required[str]
    fullname : NotRequired[str]
```

Cette version de Python ajoute un nouveau type `LiteralString` via la PEP 675 pour désigner une chaîne de caractère littérale, excluant donc les chaînes formées dynamiquement. Cela permet par typage de prévenir certains types d'injections.

```python
import subprocess
from typing import LiteralString

def run_shell(cmd: LiteralString):
    return subprocess.run(cmd, shell=True)

run_shell('ls -l')  # OK
run_shell(' '.join(['ls', '-l']))  # OK car uniquement formé de
    chaînes littérales
run_shell(input())  # KO car risque d'injection
```

Il devient aussi maintenant possible d'utiliser des types génériques avec un nombre variable d'arguments grâce à la PEP 646 qui introduit les `TypeVarTuple`. Cela vient compléter les `TypeVar` (PEP 484) qui existaient depuis Python 3.5 et permettaient de paramétrer des types génériques dans les annotations.
Cette fonctionnalité sera notamment utile pour les bibliothèques scientifiques qui peuvent nécessiter de définir une forme (*shape*) pour leurs tableaux de données, je vous renvoie aux exemples de la PEP pour mieux comprendre ce qu'il en est.

La PEP 681 apporte un nouveau décorateur, `dataclass_transform`, qui permet de décrire qu'un type se comporte comme une `dataclass`.

On note aussi l'ajout d'un module `wsgiref.types` pour spécifier les types des fonctions WSGI.

Enfin la PEP 563 initialement prévue pour Python 3.10 qui devait apporter l'évaluation tardive des annotations est à nouveau repoussée à une date indéterminée. La fonctionnalité reste toutefois utilisable à l'aide de l'import `from __future__ import annotations`.

7. Autres changements

7.1. Nouveautés

D'autres changements ont été apportés à Python ou à divers modules de la bibliothèque standard, parmi eux on retrouve :

— Les fonctions `exp2` (exponentielle de base 2) et `cbrt` (racine cubique) ajoutées au module `math`

```
>>> import math
>>> math.exp2(5)
32.0
>>> math.cbrt(64)
4.0
```

— Un alias `UTC` dans le module `datetime` pour pointer directement vers `datetime.timezone.utc`

```
>>> import datetime
>>> datetime.UTC
datetime.timezone.utc
```

— Un nouvel opérateur (`call`) dans le module `operator` pour appeler une fonction donnée en premier argument.

```
>>> import operator
>>> operator.call(print, 1, 'foo', [], sep='-')
1-foo-[]
```

— Le module `contextlib` embarque maintenant un gestionnaire de contexte `chdir` pour changer temporairement de répertoire. Attention cependant, celui-ci ne fonctionne correctement que sur un seul processus et ne gère pas le parallélisme.

```
>>> import contextlib
>>> import os
>>> os.getcwd()
'/home/entwanne'
>>> with contextlib.chdir('/tmp'):
...     os.getcwd()
...
'/tmp'
>>> os.getcwd()
'/home/entwanne'
```

— Le décorateur `singledispatch` du module `functools` supporte maintenant les unions de types.
— Ajout d'une option `-P` à l'interpréteur pour lancer Python sans ajouter le chemin courant à la variable `sys.path`.

7.2. Dépréciations

Et quelques dépréciations qui viennent avec Python 3.11 :

- Il est maintenant déconseillé de chaîner les décorateurs `classmethod` et `property` car cette fonctionnalité n'était pas très fiable.
- Le module `2to3` qui permettait de convertir du code Python 2 en Python 3 est aujourd'hui déprécié.
- Le module `binhex` et la fonction `asyncio.coroutine` précédemment dépréciées sont supprimées dans cette version.
- De nombreux modules inutilisés de la bibliothèque standard sont dépréciés et prévus pour être supprimés en Python 3.13 : `aifc`, `audioop`, `cgi`, `cgitb`, `chunk`, `crypt`, `imghdr`, `mailcap`, `msilib`, `nis`, `nntplib`, `ossaudiodev`, `pipes`, `sndhdr`, `spwd`, `sunau`, `telnetlib`, `uu` et `xdrlib`.

Conclusion

L'ensemble des changements apportés par Python 3.11 sont bien sûr à retrouver dans la documentation officielle et cette version est d'ores et déjà disponible au téléchargement sur le site officiel ou via votre gestionnaire de paquets.

La prochaine version (3.12) est prévue pour le 2 octobre prochain.

Pour toutes questions, profitez de l'espace de commentaires ci-dessous ou du forum de Zeste de Savoir.

L'image de l'article est tirée de la page release Python 3.11.0.

Liste des abréviations

JIT Just In Time. 2

TOML Tom's Obvious, Minimal Language. 9, 10

WSGI Web Server Gateway Interface. 17