
Bonnes pratiques concernant les annotations

Version 3.14.0rc3

Guido van Rossum and the Python development team

octobre 01, 2025

Python Software Foundation
Email : docs@python.org

Table des matières

1	Accès au dictionnaire des annotations d'un objet dans Python 3.10 et plus récent	2
2	Accès au dictionnaire des annotations d'un objet en Python 3.9 et antérieur	2
3	Conversion manuelle des annotations contenues dans une chaîne de caractères	3
4	Bonnes pratiques pour <code>__annotations__</code> dans toutes les versions de Python	3
5	Les curiosités concernant <code>__annotations__</code>	4
	Index	5

auteur

Larry Hastings

Résumé

Ce document a pour but de regrouper les bonnes pratiques de travail avec le dictionnaire d'annotations. Si vous écrivez du code Python qui examine les `__annotations__` des objets, nous vous encourageons à suivre les recommandations décrites dans la suite.

Ce document est organisé en quatre sections : bonnes pratiques pour accéder aux annotations d'un objet en Python 3.10 et plus récent, bonnes pratiques pour accéder aux annotations d'un objet en Python 3.9 et antérieur, autres bonnes pratiques pour `__annotations__` à appliquer quelle que soit la version de Python, et enfin les curiosités concernant `__annotations__`.

Notez que ce document traite spécifiquement du traitement des `__annotations__`, et non de l'utilisation des annotations. Si vous cherchez des informations sur la façon d'utiliser les « indications de type » dans votre code, veuillez consulter le module `typing`.

1 Accès au dictionnaire des annotations d'un objet dans Python 3.10 et plus récent

Python 3.10 adds a new function to the standard library : `inspect.get_annotations()`. In Python versions 3.10 through 3.13, calling this function is the best practice for accessing the annotations dict of any object that supports annotations. This function can also "un-stringize" stringized annotations for you.

In Python 3.14, there is a new `annotationlib` module with functionality for working with annotations. This includes a `annotationlib.get_annotations()` function, which supersedes `inspect.get_annotations()`.

Si pour une raison quelconque, `inspect.get_annotations()` n'est pas viable pour votre cas d'utilisation, vous pouvez accéder à l'attribut de données `__annotations__` manuellement. La bonne pratique pour cela a également changé en Python 3.10 : à partir de Python 3.10, le fonctionnement de `o.__annotations__` est *toujours* garanti sur les fonctions, classes et modules Python. Si vous êtes certain que l'objet que vous examinez est l'un de ces trois objets *spécifiques*, vous pouvez simplement utiliser `o.__annotations__` pour accéder au dictionnaire d'annotations de l'objet.

Cependant, d'autres types d'objets appelables – par exemple, les objets appelables créés par `functools.partial()` – peuvent ne pas avoir d'attribut `__annotations__` défini. Pour accéder à l'attribut `__annotations__` d'un objet éventuellement inconnu, la meilleure pratique, à partir de la version 3.10 de Python, est d'appeler `getattr()` avec trois arguments, par exemple `getattr(o, '__annotations__', None)`.

Dans les versions antérieures à Python 3.10, l'accès aux `__annotations__` d'une classe qui n'a pas d'annotation mais dont un parent de cette classe en a, aurait renvoyé les `__annotations__` de la classe parent. Dans les versions 3.10 et plus récentes, le résultat d'annotations de la classe enfant est un dictionnaire vide.

2 Accès au dictionnaire des annotations d'un objet en Python 3.9 et antérieur

En Python 3.9 et antérieur, l'accès au dictionnaire des annotations d'un objet est beaucoup plus compliqué que dans les versions plus récentes. Le problème est dû à un défaut de conception de ces anciennes versions de Python, notamment en ce qui concerne les annotations de classe.

La bonne pratique pour accéder au dictionnaire d'annotations d'autres objets – fonctions, autres objets appelables et modules – est la même que pour la 3.10, en supposant que vous n'appellez pas `inspect.get_annotations()` : vous devez utiliser la forme à trois arguments de `getattr()` pour accéder à l'attribut `__annotations__` de l'objet.

Malheureusement, ce n'est pas la bonne pratique pour les classes. Le problème est que, puisque `__annotations__` est optionnel sur les classes et que les classes peuvent hériter des attributs de leurs classes de base, accéder à l'attribut `__annotations__` d'une classe peut par inadvertance renvoyer le dictionnaire d'annotations d'une *classe de base*. Par exemple :

```
class Base:
    a: int = 3
    b: str = 'abc'

class Derived(Base):
    pass

print(Derived.__annotations__)
```

Ceci affiche le dictionnaire d'annotations de `Base`, pas de `Derived`.

Your code will have to have a separate code path if the object you're examining is a class (`isinstance(o, type)`). In that case, best practice relies on an implementation detail of Python 3.9 and before : if a class has annotations defined, they are stored in the class's `__dict__` dictionary. Since the class may or may not have annotations defined, best practice is to call the `get()` method on the class dict.

Pour résumer, voici un exemple de code qui accède en toute sécurité à l'attribut `__annotations__` d'un objet quelconque en Python 3.9 et antérieur :

```
if isinstance(o, type):
    ann = o.__dict__.get('__annotations__', None)
else:
    ann = getattr(o, '__annotations__', None)
```

Après avoir exécuté ce code, `ann` pourra être soit un dictionnaire, soit `None`. Nous vous conseillons de vérifier le type de `ann` en utilisant `isinstance()` avant de poursuivre l'analyse.

Note that some exotic or malformed type objects may not have a `__dict__` attribute, so for extra safety you may also wish to use `getattr()` to access `__dict__`.

3 Conversion manuelle des annotations contenues dans une chaîne de caractères

Dans les situations où certaines annotations peuvent se présenter sous forme de chaînes de caractères brutes, et que vous souhaitez évaluer ces chaînes pour trouver les valeurs Python qu'elles représentent, il est vraiment préférable d'appeler `inspect.get_annotations()` pour faire ce travail à votre place.

Si vous utilisez Python 3.9 ou antérieur ou, si pour une raison quelconque, vous ne pouvez pas utiliser `inspect.get_annotations()`, vous devrez dupliquer son principe de fonctionnement. Nous vous encourageons à examiner l'implémentation de `inspect.get_annotations()` dans la version actuelle de Python et à suivre une approche similaire.

En bref, si vous souhaitez évaluer une annotation empaquetée en chaîne de caractères sur un objet arbitraire `o` :

- Si `o` est un module, utilisez `o.__dict__` pour accéder aux noms globals lors de l'appel à `eval()`.
- Si `o` est une classe, utilisez `sys.modules[o.__module__].__dict__` pour désigner les noms globals, et `dict(vars(o))` pour désigner les locals, lorsque vous appelez `eval()`.
- Si `o` est un callable encapsulé à l'aide de `functools.update_wrapper()`, `functools.wraps()`, ou `functools.partial()`, dés-encapsulez-le itérativement en accédant à `o.__wrapped__` ou `o.func` selon le cas, jusqu'à ce que vous ayez trouvé la fonction racine.
- If `o` is a callable (but not a class), use `o.__globals__` as the globals when calling `eval()`.

Cependant, toutes les valeurs de chaîne utilisées comme annotations ne peuvent pas être transformées avec succès en valeurs Python par `eval()`. Les valeurs de chaîne peuvent théoriquement contenir des chaînes valides et, en pratique, il existe des cas d'utilisation valables pour les indications de type qui nécessitent d'annoter avec des valeurs de chaîne qui *ne peuvent pas* être évaluées. Par exemple :

- Les types d'union de style **PEP 604** avec `|`, avant que cette prise en charge ne soit ajoutée à Python 3.10.
- Les définitions qui ne sont pas nécessaires à l'exécution, importées uniquement lorsque `typing.TYPE_CHECKING` est vrai.

Si `eval()` tente d'évaluer de telles valeurs, elle échoue et lève une exception. Ainsi, lors de la conception d'une API de bibliothèque fonctionnant avec des annotations, il est recommandé de ne tenter d'évaluer des valeurs de type chaîne que si l'appelant le demande explicitement.

4 Bonnes pratiques pour `__annotations__` dans toutes les versions de Python

- Évitez d'assigner directement des objets à l'élément `__annotations__`. Laissez Python gérer le paramétrage de `__annotations__`.
- Si vous assignez directement à l'élément `__annotations__` d'un objet, vous devez toujours le définir comme un dict.
- You should avoid accessing `__annotations__` directly on any object. Instead, use `annotationlib.get_annotations()` (Python 3.14+) or `inspect.get_annotations()` (Python 3.10+).
- If you do directly access the `__annotations__` member of an object, you should ensure that it's a dictionary before attempting to examine its contents.
- Évitez de modifier les dictionnaires `__annotations__`.
- Évitez de supprimer l'attribut `__annotations__` d'un objet.

5 Les curiosités concernant `__annotations__`

Dans toutes les versions de Python 3, les fonctions créent paresseusement un dictionnaire d'annotations si aucune annotation n'est définie sur cet objet. Vous pouvez supprimer l'attribut `__annotations__` en utilisant `del fn.__annotations__`, mais si vous accédez ensuite à `fn.__annotations__`, l'objet créera un nouveau dictionnaire vide qu'il stockera et renverra quand on lui demande ses annotations. Si vous supprimez les annotations d'une fonction avant qu'elle n'ait créé paresseusement son dictionnaire d'annotations, vous obtiendrez une exception `AttributeError`; si vous utilisez `del fn.__annotations__` deux fois de suite, vous êtes sûr de toujours obtenir `AttributeError`.

Tout ce qui est dit dans le paragraphe précédent s'applique également aux objets de type classe et module en Python 3.10 et suivants.

Dans toutes les versions de Python 3, vous pouvez définir à `None` l'élément `__annotations__` sur un objet fonction. Cependant, si vous accédez ensuite aux annotations de cet objet en utilisant `fn.__annotations__`, un dictionnaire vide sera créé paresseusement, comme indiqué dans le premier paragraphe de cette section. Ce *n'est pas* le cas des modules et des classes, quelle que soit la version de Python; ces objets permettent de définir `__annotations__` à n'importe quelle valeur Python, et conserveront la valeur définie.

Si Python convertit vos annotations en chaînes de caractères (en utilisant `from __future__ import annotations`), et que vous spécifiez une chaîne de caractères comme annotation, la chaîne sera elle-même entre guillemets. En fait, l'annotation est mise entre guillemets *deux fois*. Par exemple :

```
from __future__ import annotations
def foo(a: "str"): pass

print(foo.__annotations__)
```

Ceci renvoie : `{'a': '"str"'}`. Cela ne devrait pas vraiment être considéré comme une « bizarrerie »; nous le mentionnons ici simplement parce que cela peut être surprenant.

If you use a class with a custom metaclass and access `__annotations__` on the class, you may observe unexpected behavior; see 749 for some examples. You can avoid these quirks by using `annotationlib.get_annotations()` on Python 3.14+ or `inspect.get_annotations()` on Python 3.10+. On earlier versions of Python, you can avoid these bugs by accessing the annotations from the class's `__dict__` (for example, `cls.__dict__.get('__annotations__', None)`).

In some versions of Python, instances of classes may have an `__annotations__` attribute. However, this is not supported functionality. If you need the annotations of an instance, you can use `type()` to access its class (for example, `annotationlib.get_annotations(type(myinstance))` on Python 3.14+).

Index

P

Python Enhancement Proposals

PEP 604, [3](#)

PEP 749#[pep749-metaclasses](#), [4](#)