
Boas práticas para anotações

Release 3.11.13

Guido van Rossum and the Python development team

julho 07, 2025

Python Software Foundation
Email: docs@python.org

Sumário

1	Acessando o dicionário de anotações de um objeto no Python 3.10 e nas versões mais recentes	2
2	Acessando o dicionário de anotações de um objeto no Python 3.9 e nas versões mais antigas	2
3	Recuperando manualmente anotações transformadas em strings	3
4	Melhores práticas para <code>__annotations__</code> em qualquer versão Python	3
5	Peculiaridades de <code>__annotations__</code>	4
	Índice	5

autor

Larry Hastings

Resumo

Este documento foi projetado para encapsular as melhores práticas para trabalhar com anotações. Se você escrever um código Python que examina `__annotations__` nos objetos Python, nós o encorajamos a seguir as diretrizes descritas abaixo.

Este documento está dividido em quatro seções: melhores práticas para acessar as anotações de um objeto no Python na versão 3.10 e versões mais recente, melhores práticas para acessar as anotações de um objeto no Python na versão 3.9 e versões mais antiga, outras melhores práticas para `__annotations__` para qualquer versão do Python e peculiaridades do `__annotations__`.

Note que este documento é específico sobre trabalhar com `__annotations__`, não para o uso *de* anotações. Se você está procurando por informações sobre como usar “type hints” no seu código, por favor veja o módulo `typing`

1 Acessando o dicionário de anotações de um objeto no Python 3.10 e nas versões mais recentes

O Python 3.10 adicionou uma nova função para a biblioteca padrão: `inspect.get_annotations()`. No Python 3.10 e nas versões mais recentes, chamar esta função é a melhor prática para acessar o dicionário de anotações de qualquer objeto com suporte a anotações. Esta função pode até “destextualizar” anotações textualizadas para você.

Se por alguma razão `inspect.get_annotations()` não for viável para o seu caso de uso, você pode acessar o membro de dados `__annotations__` manualmente. As melhores práticas para isto também mudaram no Python 3.10: a partir do Python 3.10, o `__annotations__` é garantido *sempre* funcionar em funções, classes e módulos Python. Se você tem certeza que o objeto que você está examinando é um desses três *exatos* objetos, pode simplesmente usar o `__annotations__` para chegar no dicionário de anotações do objeto.

Contudo, outros tipos de chamáveis – por exemplo, chamáveis criados por `functools.partial()` – podem não ter um atributo `__annotations__` definido. Ao acessar o `__annotations__` de um objeto possivelmente desconhecido, as melhores práticas nas versões de Python 3.10 e mais novas é chamar `getattr()` com três argumentos, por exemplo `getattr(o, '__annotations__', None)`.

Antes de Python 3.10, acessar `__annotations__` numa classe que não define anotações mas que possui uma classe base com anotações retorna o `__annotations__` da classe pai. A partir do Python 3.10, a anotação da classe filha será um dicionário vazio.

2 Acessando o dicionário de anotações de um objeto no Python 3.9 e nas versões mais antigas

Em Python 3.9 e versões mais antigas, acessar o dicionário de anotações de um objeto é muito mais complicado que em versões mais novas. O problema é uma falha de design em versões mais antigas do Python, especificamente a respeito de anotações de classe.

As melhores práticas para acessar os dicionários de anotações de outros objetos, funções, outros chamáveis e módulos – são as mesmas melhores práticas para 3.10, supondo que você não esteja chamando `inspect.get_annotations()`: você deve usar a função `getattr()` com três argumentos para acessar os atributos do objeto `__annotations__`.

Infelizmente, essas não são as melhores práticas para classes. O problema é que, como `__annotations__` é opcional nas classes, e posto que classes podem herdar atributos das suas classes base, acessar o atributo `__annotations__` de uma classe pode inesperadamente retornar o dicionário de anotações de uma *classe base*. Por exemplo:

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

class Derived(Base):
    pass

print(Derived.__annotations__)
```

Isso mostrará o dicionário de anotações de `Base`, não de `Derived`.

Seu código deve seguir uma abordagem diferente caso o objeto que você esteja examinando seja uma classe (`isinstance(o, type)`). Nesse caso, a melhor prática está ligada a um detalhe de implementação do Python 3.9 e anteriores: se a classe possui anotações definidas, elas são armazenadas no dicionário `__dict__` da classe. Como a classe pode ou não ter anotações definidas, a melhor prática é chamar o método `get` no dicionário da classe.

Considerando tudo isso, aqui está um exemplo de código que acessa de forma segura o atributo `__annotations__` em um objeto arbitrário em Python 3.9 e anteriores:

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

Após executar este código, `ann` deve ser um dicionário ou `None`. Você é encorajado a fazer uma checagem dupla do tipo de `ann` utilizando `isinstance()` antes de uma análise mais aprofundada.

Note que alguns tipos de objetos exóticos ou mal formatados podem não possuir um atributo `__dict__`, então por precaução você também pode querer usar `getattr()` para acessar `__dict__`.

3 Recuperando manualmente anotações transformadas em strings

Em situações em que as anotações podem ter sido transformadas em strings, e caso queira avaliar essas strings para produzir os valores de Python que elas representam, é melhor chamar `inspect.get_annotations()` para fazer isto por você.

Se estiver usando Python 3.9 ou anterior, ou por algum motivo não puder usar `inspect.get_annotations()`, será necessário replicar sua lógica. Considere examinar a implementação de `inspect.get_annotations()` na versão de Python atual e seguir uma abordagem similar.

Resumindo, caso deseje avaliar uma anotação transformada em string em um objeto arbitrário `o`:

- Se `o` for um módulo, use `o.__dict__` como `globals` ao chamar `eval()`.
- Se `o` for uma classe, use `sys.modules[o.__module__].__dict__` como `globals`, e `dict(vars(o))` como `locals`, quando chamar `eval()`.
- Se `o` for um chamável envolto em um invólucro usando `functools.update_wrapper()`, `functools.wraps()` ou `functools.partial()`, desenvolva-o iterativamente acessando `o.__wrapped__` ou `o.func` conforme apropriado, até encontrar a função raiz.
- Caso `o` seja chamável (mas não uma classe), use `o.__globals__` como `globals` quando chamar `eval()`.

Contudo, nem todas strings usadas como anotações podem ser convertidas em valores de Python utilizando `eval()`. Valores de string poderiam teoricamente conter qualquer string válida, e na prática existem casos válidos para dicas de tipo que requerem anotar com valores de strings que *não* podem ser executados. Por exemplo:

- Tipos de união **PEP 604** usando `|`, antes do suporte a isso ser adicionado em Python 3.10.
- Definições que não são necessárias no ambiente de execução, apenas importadas quando `typing.TYPE_CHECKING` é verdadeiro.

Caso `eval()` tente executar tais valores, levantará uma exceção. Então, quando projetar uma biblioteca API que trabalha com anotações, é recomendado que tente executar os valores das strings apenas quando for solicitado explicitamente.

4 Melhores práticas para `__annotations__` em qualquer versão Python

- Evite atribuir diretamente ao membro `__annotations__` dos objetos. Deixe que Python gerenciar a configuração de `__annotations__`.
- Se você atribuir diretamente ao membro `__annotations__` do objeto, sempre o defina como um objeto `dict`.
- Caso acesse diretamente o membro `__annotations__` de um objeto, certifique-se de que ele é um dicionário antes de tentar examinar seu conteúdo.
- Evite modificar o dicionário `__annotations__`.

- Evite deletar o atributo `__annotations__` de um objeto.

5 Peculiaridades de `__annotations__`

Em todas as versões de Python 3, objetos de função criam preguiçosamente um dicionário de anotações caso nenhuma anotação seja definida nesse objeto. Você pode deletar o atributo `__annotations__` utilizando `del fn.__annotations__`, mas ao acessar `fn.__annotations__` o objeto criará um novo dicionário vazio que será armazenado e retornado como suas anotações. Apagar as anotações de uma função antes que ela tenha criado preguiçosamente seu dicionário de anotações irá produzir um `AttributeError`; ao utilizar `del fn.__annotations__` duas vezes em sequência, é garantido que será produzido um `AttributeError`.

O citado no parágrafo acima também se aplica para classes e objetos de módulo em Python 3.10 e posteriores.

Em todas as versões de Python 3, você pode definir `__annotations__` como `None` em um objeto de função. Contudo, acessar em sequência as anotações nesse objeto utilizando `fn.__annotations__` irá criar preguiçosamente um dicionário vazio como descrito no primeiro parágrafo dessa seção. Isso *não* é válido para módulos e classes, em qualquer versão de Python; esses objetos permitem atribuir `__annotations__` a qualquer valor de Python, e armazenam qualquer valor atribuído.

Se Python transformar sua anotação em string (utilizando `from __future__ import annotations`), e você especificar uma string como anotação, essa string será posta entre aspas. Na prática a anotação receberá aspas *duplas*. Por exemplo:

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

print(foo.__annotations__)
```

Isso exibe `{'a': "'str'"}`. Não considere isso como uma “peculiaridade”; foi mencionado aqui simplesmente porque pode ser surpreendente.

Índice

P

Propostas Estendidas Python
PEP 604,[3](#)