
Prácticas recomendadas para las anotaciones

Versión 3.10.16

Guido van Rossum
and the Python development team

diciembre 07, 2024

Python Software Foundation
Email: docs@python.org

Índice general

| | | |
|---|--|---|
| 1 | Acceder al diccionario de anotaciones de un objeto en las versiones de Python 3.10 y posteriores | 2 |
| 2 | Acceder al diccionario de anotaciones de un objeto en las versiones de Python 3.9 y anteriores | 2 |
| 3 | Desencadenamiento manual de anotaciones en cadena | 3 |
| 4 | Prácticas recomendadas para <code>__annotations__</code> en cualquier versión de Python | 4 |
| 5 | Peculiaridades de <code>__annotations__</code> | 4 |
| | Índice | 5 |

autor Larry Hastings

Resumen

Este documento se designó para encapsular las prácticas recomendadas para trabajar con diccionarios de anotaciones. Si escribe código Python que examina `__annotations__` en objetos Python, recomendamos seguir las pautas que se describen a continuación.

El documento se organizó en 4 secciones: prácticas recomendadas para acceder a las anotaciones de un objeto en las versiones de Python 3.10 y posteriores, prácticas recomendadas para acceder a las anotaciones de un objeto en las versiones de Python 3.9 y anteriores, otras prácticas recomendadas para `__annotations__` que aplican a cualquier versión de Python y peculiaridades de `__annotations__`.

Tome en cuenta que este documento trabaja específicamente con `__annotations__`, no usarlo *para* anotaciones. Si está buscando información sobre cómo usar «anotaciones de tipado» en su código, consulte el módulo `typing`.

1 Acceder al diccionario de anotaciones de un objeto en las versiones de Python 3.10 y posteriores

Python 3.10 agrega una nueva función a la biblioteca estándar: `inspect.get_annotations()`. En las versiones de Python 3.10 y posteriores, llamar esta función es la mejor práctica para acceder al diccionario de anotaciones de cualquier objeto que admita anotaciones. También esta función puede «desencadenar» anotaciones en cadena.

Si por alguna razón `inspect.get_annotations()` no es viable para su caso de uso, puede acceder manualmente al miembro de datos `__annotations__`. La práctica recomendada para esto también cambió en Python 3.10: a partir de Python 3.10, se garantiza que `o.__annotations__` siempre opere en funciones, clases y módulos de Python. Si está seguro de que el objeto que está examinando es uno de los tres objetos *específicos*, puede usar simplemente `o.__annotations__` para obtener el diccionario de anotaciones del objeto.

Sin embargo, otros tipos de llamadas – por ejemplo, invocables creados por `functools.partial()` – pueden no tener definido un atributo `__annotations__`. Al acceder a `__annotations__` de un objeto posiblemente desconocido, la práctica recomendada de las versiones de Python 3.10 y posteriores es llamar `getattr()` con tres argumentos, por ejemplo `getattr(o, '__annotations__', None)`.

Before Python 3.10, accessing `__annotations__` on a class that defines no annotations but that has a parent class with annotations would return the parent's `__annotations__`. In Python 3.10 and newer, the child class's annotations will be an empty dict instead.

2 Acceder al diccionario de anotaciones de un objeto en las versiones de Python 3.9 y anteriores

En versiones de Python 3.9 y anteriores, acceder al diccionario de anotaciones de un objeto es mucho más complicado que en versiones más recientes. El problema es un defecto de diseño en estas versiones anteriores de Python, específicamente relacionado con las anotaciones de clase.

La práctica recomendada para acceder al diccionario de anotaciones de otros objetos – funciones, otros invocables y módulos – es la misma que la de la 3.10, asumiendo que no está llamando `inspect.get_annotations()`: debe usar tres argumentos de `getattr()` para acceder al atributo `__annotations__` del objeto.

Desafortunadamente, esta no es la mejor práctica para las clases. El problema es que, dado que `__annotations__` es opcional en las clases, y debido a que las clases pueden heredar atributos desde sus clases base, acceder al atributo `__annotations__` de una clase puede retornar por inadvertencia el diccionario de anotaciones de una *clase base*. Como ejemplo:

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

class Derived(Base):
    pass

print(Derived.__annotations__)
```

Esto imprimirá el diccionario de anotaciones de `Base`, no de `Derived`.

Su código deberá tener una ruta de código separada si el objeto que está examinando es una clase (`isinstance(o, type)`). En este caso, la práctica recomendada se basa en un detalle de implementación de las versiones de Python 3.9 y anteriores: si una clase tiene anotaciones definidas, se almacenan en el diccionario `__dict__` de la clase. Dado que la clase puede o no tener anotaciones definidas, la mejor práctica es llamar al método `get` en el diccionario de la clase.

Para ponerlo todo junto, aquí hay un código de muestra que accede de forma segura al atributo `__annotations__` en un objeto arbitrario en las versiones de Python 3.9 y anteriores:

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

Después de ejecutar este código, `ann` debería ser un diccionario o `None`. Recomendamos que vuelva a verificar el tipo de `ann` usando `isinstance()` antes de un examen más detenido.

Tome en cuenta que algunos objetos de tipo exótico o con formato incorrecto pueden no tener un atributo `__dict__`, así que para mayor seguridad, también puede usar `getattr()` para acceder a `__dict__`.

3 Desencadenamiento manual de anotaciones en cadena

En situaciones donde algunas anotaciones pueden estar «encadenadas», y desea evaluar esas cadenas de caracteres para producir los valores de Python que representan, lo mejor es llamar a `inspect.get_annotations()` para que haga este trabajo por usted.

Si está usando la versión de Python 3.9 o anterior, o si por alguna razón no puede usar `inspect.get_annotations()`, necesitará duplicar su lógica. Recomendamos que examine la implementación de `inspect.get_annotations()` en la versión actual de Python y siga un enfoque similar.

En pocas palabras, si desea evaluar una anotación en cadena en un objeto arbitrario `o`:

- Si `o` es un módulo, use `o.__dict__` como `globals` cuando llame a `eval()`.
- Si `o` es una clase, use `sys.modules[o.__module__].__dict__` como `globals` y `dict(vars(o))` como `locals` cuando llame a `eval()`.
- Si `o` es un invocable envuelto usando `functools.update_wrapper()`, `functools.wraps()` o `functools.partial()`, lo desenvuelve iterativamente accediendo a `o.__wrapped__` o `o.func` según corresponda, hasta que haya encontrado la función raíz sin envolver.
- Si `o` es un invocable (pero no una clase), use `o.__globals__` como `globals` cuando llame a `eval()`.

Sin embargo, no todos los valores de cadena de caracteres usados como anotaciones se pueden convertir correctamente en valores de Python mediante `eval()`. Los valores de cadena de caracteres pueden contener, teóricamente, cualquier cadena de caracteres válida, y en la práctica, hay varios casos de uso válidos para anotaciones de tipo que requieren anotaciones con valores de cadena de caracteres que *no pueden* evaluarse específicamente. Por ejemplo:

- **PEP 604** union types using `|`, before support for this was added to Python 3.10.
- Las definiciones que no son necesarias en tiempo de ejecución, sólo se importan cuando `typing.TYPE_CHECKING` es verdadero.

Si `eval()` intenta evaluar tales valores, fallará y lanzará una excepción. Por lo tanto, al diseñar una API de biblioteca que funcione con anotaciones, se recomienda sólo intentar evaluar valores de cadena de caracteres cuando la llamada lo solicite explícitamente.

4 Prácticas recomendadas para `__annotations__` en cualquier versión de Python

- Debe evitar asignar directamente al miembro `__annotations__` de objetos. Deje que Python administre la configuración `__annotations__`.
- Si asigna directamente al miembro `__annotations__` de un objeto, siempre debe establecerlo en un objeto `dict`.
- Si accede directamente al miembro `__annotations__` de un objeto, debe asegurarse de que sea un diccionario antes de intentar examinar su contenido.
- Debe evitar modificar los diccionarios `__annotations__`.
- Debe evitar eliminar el atributo `__annotations__` de un objeto.

5 Peculiaridades de `__annotations__`

En todas las versiones de Python 3, los objetos de función crean de forma diferida un diccionario de anotaciones si no hay anotaciones definidas en ese objeto. Puede eliminar el atributo `__annotations__` usando `del fn.__annotations__`, pero si luego accede a `fn.__annotations__`, el objeto creará un diccionario nuevo vacío que almacenará y retornará como sus anotaciones. Eliminar las anotaciones en una función antes de que haya creado su diccionario de anotaciones de forma diferida arrojará un `AttributeError`; el uso de dos veces seguidas de `del fn.__annotations__` garantiza que siempre arroje un `AttributeError`.

Todo en el párrafo anterior también se aplica a los objetos de clase y módulo en las versiones de Python 3.10 y posteriores.

En todas las versiones de Python 3, puede establecer `__annotations__` en un objeto de función en `None`. Sin embargo, al acceder después a las anotaciones en ese objeto usando `fn.__annotations__` se creará un diccionario vacío de forma diferida según el primer párrafo de esta sección. Esto *no* es cierto para módulos y clases, en cualquier versión de Python; esos objetos permiten establecer `__annotations__` en cualquier valor de Python, y conservarán cualquier valor que se establezca.

Si Python encadena sus anotaciones por usted (usando `from __future__ import annotations`), y especifica una cadena de caracteres como una anotación, la cadena de caracteres en sí se citará. En efecto, la anotación se cita *dos veces*. Por ejemplo:

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

print(foo.__annotations__)
```

Esto imprime `{'a': "'str'"}`. En realidad, esto no debería considerarse una «peculiaridad»; aquí se menciona simplemente porque podría sorprenderle.

Índice

P

Python Enhancement Proposals
PEP 604, [3](#)