Annotations Best Practices

Release 3.13.0rc1

Guido van Rossum and the Python development team

September 07, 2024

Python Software Foundation
Email: docs@python.org

Contents

1 Accessing The Annotations Dict Of An Object In Python 3.10 And Newer 2

2 Accessing The Annotations Dict Of An Object In Python 3.9 And Older 2

3 Manually Un-Stringizing Stringized Annotations 3

4 Best Practices For `__annotations__` In Any Python Version 3

5 `__annotations__` Quirks 4

Index 5

author
Larry Hastings

Abstract

This document is designed to encapsulate the best practices for working with annotations dicts. If you write Python code that examines `__annotations__` on Python objects, we encourage you to follow the guidelines described below.

The document is organized into four sections: best practices for accessing the annotations of an object in Python versions 3.10 and newer, best practices for accessing the annotations of an object in Python versions 3.9 and older, other best practices for `__annotations__` that apply to any Python version, and quirks of `__annotations__`.

Note that this document is specifically about working with `__annotations__`, not uses *for* annotations. If you're looking for information on how to use "type hints" in your code, please see the `typing` module.

1 Accessing The Annotations Dict Of An Object In Python 3.10 And Newer

Python 3.10 adds a new function to the standard library: `inspect.get_annotations()`. In Python versions 3.10 and newer, 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.

If for some reason `inspect.get_annotations()` isn't viable for your use case, you may access the `__annotations__` data member manually. Best practice for this changed in Python 3.10 as well: as of Python 3.10, `o.__annotations__` is guaranteed to *always* work on Python functions, classes, and modules. If you're certain the object you're examining is one of these three *specific* objects, you may simply use `o.__annotations__` to get at the object's annotations dict.

However, other types of callables–for example, callables created by `functools.partial()`–may not have an `__annotations__` attribute defined. When accessing the `__annotations__` of a possibly unknown object, best practice in Python versions 3.10 and newer is to call `getattr()` with three arguments, for example `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 Accessing The Annotations Dict Of An Object In Python 3.9 And Older

In Python 3.9 and older, accessing the annotations dict of an object is much more complicated than in newer versions. The problem is a design flaw in these older versions of Python, specifically to do with class annotations.

Best practice for accessing the annotations dict of other objects–functions, other callables, and modules–is the same as best practice for 3.10, assuming you aren't calling `inspect.get_annotations()`: you should use three-argument `getattr()` to access the object's `__annotations__` attribute.

Unfortunately, this isn't best practice for classes. The problem is that, since `__annotations__` is optional on classes, and because classes can inherit attributes from their base classes, accessing the `__annotations__` attribute of a class may inadvertently return the annotations dict of a *base class.* As an example:

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

class Derived(Base):
    pass

print(Derived.__annotations__)
```

This will print the annotations dict from `Base`, not `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.

To put it all together, here is some sample code that safely accesses the `__annotations__` attribute on an arbitrary object in Python 3.9 and before:

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

After running this code, `ann` should be either a dictionary or `None`. You're encouraged to double-check the type of `ann` using `isinstance()` before further examination.

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 Manually Un-Stringizing Stringized Annotations

In situations where some annotations may be "stringized", and you wish to evaluate those strings to produce the Python values they represent, it really is best to call `inspect.get_annotations()` to do this work for you.

If you're using Python 3.9 or older, or if for some reason you can't use `inspect.get_annotations()`, you'll need to duplicate its logic. You're encouraged to examine the implementation of `inspect.get_annotations()` in the current Python version and follow a similar approach.

In a nutshell, if you wish to evaluate a stringized annotation on an arbitrary object `o`:

- If `o` is a module, use `o.__dict__` as the `globals` when calling `eval()`.
- If `o` is a class, use `sys.modules[o.__module__].__dict__` as the `globals`, and `dict(vars(o))` as the `locals`, when calling `eval()`.
- If `o` is a wrapped callable using `functools.update_wrapper()`, `functools.wraps()`, or `functools.partial()`, iteratively unwrap it by accessing either `o.__wrapped__` or `o.func` as appropriate, until you have found the root unwrapped function.
- If `o` is a callable (but not a class), use `o.__globals__` as the globals when calling `eval()`.

However, not all string values used as annotations can be successfully turned into Python values by `eval()`. String values could theoretically contain any valid string, and in practice there are valid use cases for type hints that require annotating with string values that specifically *can't* be evaluated. For example:

- **PEP 604** union types using `|`, before support for this was added to Python 3.10.
- Definitions that aren't needed at runtime, only imported when `typing.TYPE_CHECKING` is true.

If `eval()` attempts to evaluate such values, it will fail and raise an exception. So, when designing a library API that works with annotations, it's recommended to only attempt to evaluate string values when explicitly requested to by the caller.

4 Best Practices For `__annotations__` In Any Python Version

- You should avoid assigning to the `__annotations__` member of objects directly. Let Python manage setting `__annotations__`.
- If you do assign directly to the `__annotations__` member of an object, you should always set it to a `dict` object.
- If you directly access the `__annotations__` member of an object, you should ensure that it's a dictionary before attempting to examine its contents.
- You should avoid modifying `__annotations__` dicts.

- You should avoid deleting the `__annotations__` attribute of an object.

5 `__annotations__` Quirks

In all versions of Python 3, function objects lazy-create an annotations dict if no annotations are defined on that object. You can delete the `__annotations__` attribute using `del fn.__annotations__`, but if you then access `fn.__annotations__` the object will create a new empty dict that it will store and return as its annotations. Deleting the annotations on a function before it has lazily created its annotations dict will throw an `AttributeError`; using `del fn.__annotations__` twice in a row is guaranteed to always throw an `AttributeError`.

Everything in the above paragraph also applies to class and module objects in Python 3.10 and newer.

In all versions of Python 3, you can set `__annotations__` on a function object to `None`. However, subsequently accessing the annotations on that object using `fn.__annotations__` will lazy-create an empty dictionary as per the first paragraph of this section. This is *not* true of modules and classes, in any Python version; those objects permit setting `__annotations__` to any Python value, and will retain whatever value is set.

If Python stringizes your annotations for you (using `from __future__ import annotations`), and you specify a string as an annotation, the string will itself be quoted. In effect the annotation is quoted *twice.* For example:

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

print(foo.__annotations__)
```

This prints `{'a': "'str'"}`. This shouldn't really be considered a "quirk"; it's mentioned here simply because it might be surprising.

Index

P

Python Enhancement Proposals
    PEP 604, 3