`!annotationlib` --- Functionality for introspecting annotations

- Annotation semantics
- Classes
- Functions
- Recipes
  - Using annotations in a metaclass
  - Creating a custom callable annotate function
- Limitations of the STRING format
- Limitations of the FORWARDREF format
- Security implications of introspecting annotations

annotationlib

3.14

Source code: `Lib/annotationlib.py`

default

import annotationlib from annotationlib import *

The `!annotationlib` module provides tools for introspecting `annotations <annotation>` on modules, classes, and functions.

Annotations are `lazily evaluated <lazy-evaluation>` and often contain forward references to objects that are not yet defined when the annotation is created. This module provides a set of low-level tools that can be used to retrieve annotations in a reliable way, even in the presence of forward references and other edge cases.

This module supports retrieving annotations in three main formats (see `Format`), each of which works best for different use cases:

- `~Format.VALUE` evaluates the annotations and returns their value. This is most straightforward to work with, but it may raise errors, for example if the annotations contain references to undefined names.
- `~Format.FORWARDREF` returns `ForwardRef` objects for annotations that cannot be resolved, allowing you to inspect the annotations without evaluating them. This is useful when you need to work with annotations that may contain unresolved forward references.
- `~Format.STRING` returns the annotations as a string, similar to how it would appear in the source file. This is useful for documentation generators that want to display annotations in a readable way.

The `get_annotations` function is the main entry point for retrieving annotations. Given a function, class, or module, it returns an annotations dictionary in the requested format. This module also provides functionality for working directly with the `annotate function` that is used to evaluate annotations, such as `get_annotate_from_class_namespace` and `call_annotate_function`, as well as the `call_evaluate_function` function for working with `evaluate functions <evaluate function>`.

> [!CAUTION] Most functionality in this module can execute arbitrary code; see `the security section <annotationlib-security>` for more information.

`649` proposed the current model for how annotations work in Python.

`749` expanded on various aspects of `649` and introduced the `!annotationlib` module.

`annotations-howto` provides best practices for working with annotations.

`typing-extensions` provides a backport of `get_annotations` that works on earlier versions of Python.

Annotation semantics

The way annotations are evaluated has changed over the history of Python 3, and currently still depends on a `future import <future>`. There have been execution models for annotations:

- *Stock semantics* (default in Python 3.0 through 3.13; see `3107` and `526`): Annotations are evaluated eagerly, as they are encountered in the source code.
- *Stringified annotations* (used with `from __future__ import annotations` in Python 3.7 and newer; see `563`): Annotations are stored as strings only.
- *Deferred evaluation* (default in Python 3.14 and newer; see `649` and `749`): Annotations are evaluated lazily, only when they are accessed.

As an example, consider the following program:

```
def func(a: Cls) -> None:
    print(a)

class Cls: pass

print(func.__annotations__)
```

This will behave as follows:

- Under stock semantics (Python 3.13 and earlier), it will throw a `NameError` at the line where `func` is defined, because `Cls` is an undefined name at that point.
- Under stringified annotations (if `from __future__ import annotations` is used), it will print `{'a': 'Cls', 'return': 'None'}`.
- Under deferred evaluation (Python 3.14 and later), it will print `{'a': <class 'Cls'>, 'return': None}`.

Stock semantics were used when function annotations were first introduced in Python 3.0 (by `3107`) because this was the simplest, most obvious way to implement annotations. The same execution model was used when variable annotations were introduced in Python 3.6 (by `526`). However, stock semantics caused problems when using annotations as type hints, such as a need to refer to names that are not yet defined when the annotation is encountered. In addition, there were performance problems with executing annotations at module import time. Therefore, in Python 3.7, `563` introduced the ability to store annotations as strings using the `from __future__ import annotations` syntax. The plan at the time was to eventually make this behavior the default, but a problem appeared: stringified annotations are more difficult to process for those who introspect annotations at runtime. An alternative proposal, `649`, introduced the third execution model, deferred evaluation, and was implemented in Python 3.14. Stringified annotations are still used if `from __future__ import annotations` is present, but this behavior will eventually be removed.

Classes

An `~enum.IntEnum` describing the formats in which annotations can be returned. Members of the enum, or their equivalent integer values, can be passed to `get_annotations` and other functions in this module, as well as to `~object.__annotate__` functions.

VALUE

Values are the result of evaluating the annotation expressions.

VALUE_WITH_FAKE_GLOBALS

Special value used to signal that an annotate function is being evaluated in a special environment with fake globals. When passed this value, annotate functions should either return the same value as for the `Format.VALUE` format, or raise `NotImplementedError` to signal that they do not support execution in this environment. This format is only used internally and should not be passed to the functions in this module.

FORWARDREF

Values are real annotation values (as per `Format.VALUE` format) for defined values, and `ForwardRef` proxies for undefined values. Real objects may contain references to `ForwardRef` proxy objects.

STRING

Values are the text string of the annotation as it appears in the source code, up to modifications including, but not restricted to, whitespace normalizations and constant values optimizations.

The exact values of these strings may change in future versions of Python.

3.14

A proxy object for forward references in annotations.

Instances of this class are returned when the `~Format.FORWARDREF` format is used and annotations contain a name that cannot be resolved. This can happen when a forward reference is used in an annotation, such as when a class is referenced before it is defined.

__forward_arg__

A string containing the code that was evaluated to produce the `~ForwardRef`. The string may not be exactly equivalent to the original source.

evaluate(*, owner=None, globals=None, locals=None, type_params=None, format=Format.VALUE)

Evaluate the forward reference, returning its value.

If the *format* argument is `~Format.VALUE` (the default), this method may throw an exception, such as `NameError`, if the forward reference refers to a name that cannot be resolved. The arguments to this method can be used to provide bindings for names that would otherwise be undefined. If the *format* argument is `~Format.FORWARDREF`, the method will never throw an exception, but may return a `~ForwardRef` instance. For example, if the forward reference object contains the code `list[undefined]`, where `undefined` is a name that is not defined, evaluating it with the `~Format.FORWARDREF` format will return `list[ForwardRef('undefined')]`. If the *format* argument is `~Format.STRING`, the method will return `~ForwardRef.__forward_arg__`.

The *owner* parameter provides the preferred mechanism for passing scope information to this method. The owner of a `~ForwardRef` is the object that contains the annotation from which the `~ForwardRef` derives, such as a module object, type object, or function object.

The *globals*, *locals*, and *type_params* parameters provide a more precise mechanism for influencing the names that are available when the `~ForwardRef` is evaluated. *globals* and *locals* are passed to `eval`, representing the global and local namespaces in which the name is evaluated. The *type_params* parameter is relevant for objects created using the native syntax for `generic classes <generic-classes>` and `functions <generic-functions>`. It is a tuple of `type parameters <type-params>` that are in scope while the forward reference is being evaluated. For example, if evaluating a `~ForwardRef` retrieved from an annotation found in the class namespace of a generic class `C`, *type_params* should be set to `C.__type_params__`.

`~ForwardRef` instances returned by `get_annotations` retain references to information about the scope they originated from, so calling this method with no further arguments may be sufficient to evaluate such objects. `~ForwardRef` instances created by other means may not have any information about their scope, so passing arguments to this method may be necessary to evaluate them successfully.

If no *owner*, *globals*, *locals*, or *type_params* are provided and the `~ForwardRef` does not contain information about its origin, empty globals and locals dictionaries are used.

3.14

Functions

annotations_to_string(annotations)

Convert an annotations dict containing runtime values to a dict containing only strings. If the values are not already strings, they are converted using `type_repr`. This is meant as a helper for user-provided annotate functions that support the `~Format.STRING` format but do not have access to the code creating the annotations.

For example, this is used to implement the `~Format.STRING` format for `typing.TypedDict` classes created through the functional syntax:

>>> from typing import TypedDict >>> Movie = TypedDict("movie", {"name": str, "year": int}) >>> get_annotations(Movie, format=Format.STRING) {'name': 'str', 'year': 'int'}

3.14

call_annotate_function(annotate, format, *, owner=None)

Call the `annotate function` *annotate* with the given *format*, a member of the `Format` enum, and return the annotations dictionary produced by the function.

This helper function is required because annotate functions generated by the compiler for functions, classes, and modules only support the `~Format.VALUE` format when called directly. To support other formats, this function calls the annotate function in a special environment that allows it to produce annotations in the other formats. This is a useful building block when implementing functionality that needs to partially evaluate annotations while a class is being constructed.

owner is the object that owns the annotation function, usually a function, class, or module. If provided, it is used in the `~Format.FORWARDREF` format to produce a `ForwardRef` object that carries more information.

PEP PEP 649 <649#the-stringizer-and-the-fake-globals-environment> contains an explanation of the implementation technique used by this function.

3.14

call_evaluate_function(evaluate, format, *, owner=None)

Call the `evaluate function` *evaluate* with the given *format*, a member of the `Format` enum, and return the value produced by the function. This is similar to `call_annotate_function`, but the latter always returns a dictionary mapping strings to annotations, while this function returns a single value.

This is intended for use with the evaluate functions generated for lazily evaluated elements related to type aliases and type parameters:

- `typing.TypeAliasType.evaluate_value`, the value of type aliases
- `typing.TypeVar.evaluate_bound`, the bound of type variables
- `typing.TypeVar.evaluate_constraints`, the constraints of type variables
- `typing.TypeVar.evaluate_default`, the default value of type variables
- `typing.ParamSpec.evaluate_default`, the default value of parameter specifications
- `typing.TypeVarTuple.evaluate_default`, the default value of type variable tuples

owner is the object that owns the evaluate function, such as the type alias or type variable object.

format can be used to control the format in which the value is returned:

>>> type Alias = undefined >>> call_evaluate_function(Alias.evaluate_value, Format.VALUE) Traceback (most recent call last): ... NameError: name 'undefined' is not defined >>> call_evaluate_function(Alias.evaluate_value, Format.FORWARDREF) ForwardRef('undefined') >>> call_evaluate_function(Alias.evaluate_value, Format.STRING) 'undefined'

3.14

get_annotate_from_class_namespace(namespace)

Retrieve the `annotate function` from a class namespace dictionary *namespace*. Return `!None` if the namespace does not contain an annotate function. This is primarily useful before the class has been fully created (e.g., in a metaclass); after the class exists, the annotate function can be retrieved with `cls.__annotate__`. See `below <annotationlib-metaclass>` for an example using this function in a metaclass.

3.14

get_annotations(obj, *, globals=None, locals=None, eval_str=False, format=Format.VALUE)

Compute the annotations dict for an object.

obj may be a callable, class, module, or other object with `~object.__annotate__` or `~object.__annotations__` attributes. Passing any other object raises `TypeError`.

The *format* parameter controls the format in which annotations are returned, and must be a member of the `Format` enum or its integer equivalent. The different formats work as follows:

- VALUE: `!object.__annotations__` is tried first; if that does not exist, the `!object.__annotate__` function is called if it exists.
- FORWARDREF: If `!object.__annotations__` exists and can be evaluated successfully, it is used; otherwise, the `!object.__annotate__` function is called. If it does not exist either, `!object.__annotations__` is tried again and any error from accessing it is re-raised.
  - When calling `!object.__annotate__` it is first called with `~Format.FORWARDREF`. If this is not implemented, it will then check if `~Format.VALUE_WITH_FAKE_GLOBALS` is supported and use that in the fake globals environment. If neither of these formats are supported, it will fall back to using `~Format.VALUE`. If `~Format.VALUE` fails, the error from this call will be raised.
- STRING: If `!object.__annotate__` exists, it is called first; otherwise, `!object.__annotations__` is used and stringified using `annotations_to_string`.
  - When calling `!object.__annotate__` it is first called with `~Format.STRING`. If this is not implemented, it will then check if `~Format.VALUE_WITH_FAKE_GLOBALS` is supported and use that in the fake globals environment. If neither of these formats are supported, it will fall back to using `~Format.VALUE` with the result converted using `annotations_to_string`. If `~Format.VALUE` fails, the error from this call will be raised.

Returns a dict. `!get_annotations` returns a new dict every time it's called; calling it twice on the same object will return two different but equivalent dicts.

This function handles several details for you:

- If *eval_str* is true, values of type `!str` will be un-stringized using `eval`. This is intended for use with stringized annotations (`from __future__ import annotations`). It is an error to set *eval_str* to true with formats other than `Format.VALUE`.

- If *obj* doesn't have an annotations dict, returns an empty dict. (Functions and methods always have an annotations dict; classes, modules, and other types of callables may not.)
- Ignores inherited annotations on classes, as well as annotations on metaclasses. If a class doesn't have its own annotations dict, returns an empty dict.
- All accesses to object members and dict values are done using `getattr()` and `dict.get()` for safety.

eval_str controls whether or not values of type `!str` are replaced with the result of calling `eval` on those values:

- If eval_str is true, `eval` is called on values of type `!str`. (Note that `!get_annotations` doesn't catch exceptions; if `eval` raises an exception, it will unwind the stack past the `!get_annotations` call.)
- If *eval_str* is false (the default), values of type `!str` are unchanged.

globals and *locals* are passed in to `eval`; see the documentation for `eval` for more information. If *globals* or *locals* is `!None`, this function may replace that value with a context-specific default, contingent on `type(obj)`:

- If *obj* is a module, *globals* defaults to `obj.__dict__`.
- If *obj* is a class, *globals* defaults to `sys.modules[obj.__module__].__dict__` and *locals* defaults to the *obj* class namespace.
- If *obj* is a callable, *globals* defaults to `obj.__globals__ <function.__globals__>`, although if *obj* is a wrapped function (using `functools.update_wrapper`) or a `functools.partial` object, it is unwrapped until a non-wrapped function is found.

Calling `!get_annotations` is best practice for accessing the annotations dict of any object. See `annotations-howto` for more information on annotations best practices.

>>> def f(a: int, b: str) -> float: ... pass >>> get_annotations(f) {'a': <class 'int'>, 'b': <class 'str'>, 'return': <class 'float'>}

3.14

type_repr(value)

Convert an arbitrary Python value to a format suitable for use by the `~Format.STRING` format. This calls `repr` for most objects, but has special handling for some objects, such as type objects.

This is meant as a helper for user-provided annotate functions that support the `~Format.STRING` format but do not have access to the code creating the annotations. It can also be used to provide a user-friendly string representation for other objects that contain values that are commonly encountered in annotations.

Recipes

Using annotations in a metaclass

A `metaclass <metaclasses>` may want to inspect or even modify the annotations in a class body during class creation. Doing so requires retrieving annotations from the class namespace dictionary. For classes created with `from __future__ import annotations`, the annotations will be in the `__annotations__` key of the dictionary. For other classes with annotations, `get_annotate_from_class_namespace` can be used to get the annotate function, and `call_annotate_function` can be used to call it and retrieve the annotations. Using the `~Format.FORWARDREF` format will usually be best, because this allows the annotations to refer to names that cannot yet be resolved when the class is created.

To modify the annotations, it is best to create a wrapper annotate function that calls the original annotate function, makes any necessary adjustments, and returns the result.

Below is an example of a metaclass that filters out all `typing.ClassVar` annotations from the class and puts them in a separate attribute:

```
import annotationlib
import typing

class ClassVarSeparator(type):
   def __new__(mcls, name, bases, ns):
      if "__annotations__" in ns:  # from __future__ import annotations
         annotations = ns["__annotations__"]
         classvar_keys = {
            key for key, value in annotations.items()
            # Use string comparison for simplicity; a more robust solution
            # could use annotationlib.ForwardRef.evaluate
            if value.startswith("ClassVar")
         }
         classvars = {key: annotations[key] for key in classvar_keys}
         ns["__annotations__"] = {
            key: value for key, value in annotations.items()
            if key not in classvar_keys
```

```
        }
        wrapped_annotate = None
    elif annotate := annotationlib.get_annotate_from_class_namespace(ns):
        annotations = annotationlib.call_annotate_function(
            annotate, format=annotationlib.Format.FORWARDREF
        )
        classvar_keys = {
            key for key, value in annotations.items()
            if typing.get_origin(value) is typing.ClassVar
        }
        classvars = {key: annotations[key] for key in classvar_keys}

        def wrapped_annotate(format):
            annos = annotationlib.call_annotate_function(annotate, format, owner=typ)
            return {key: value for key, value in annos.items() if key not in classvar_keys}

    else:  # no annotations
        classvars = {}
        wrapped_annotate = None
    typ = super().__new__(mcls, name, bases, ns)

    if wrapped_annotate is not None:
        # Wrap the original __annotate__ with a wrapper that removes ClassVars
        typ.__annotate__ = wrapped_annotate
    typ.classvars = classvars  # Store the ClassVars in a separate attribute
    return typ
```

Creating a custom callable annotate function

Custom `annotate functions <annotate function>` may be literal functions like those automatically generated for functions, classes, and modules. Or, they may wish to utilise the encapsulation provided by classes, in which case any `callable` can be used as an `annotate function`.

To provide the `~Format.VALUE`, `~Format.STRING`, or `~Format.FORWARDREF` formats directly, an `annotate function` must provide the following attribute:

- A callable `__call__` with signature `__call__(format, /) -> dict`, that does not raise a `NotImplementedError` when called with a supported format.

To provide the `~Format.VALUE_WITH_FAKE_GLOBALS` format, which is used to automatically generate `~Format.STRING` or `~Format.FORWARDREF` if they are not supported directly, `annotate functions <annotate function>` must provide the following attributes:

- A callable `__call__` with signature `__call__(format, /) -> dict`, that does not raise a `NotImplementedError` when called with `~Format.VALUE_WITH_FAKE_GLOBALS`.
- A `code object <code-objects>` `__code__` containing the compiled code for the annotate function.
- Optional: A tuple of the function's positional defaults `__defaults__`, if the function represented by `__code__` uses any positional defaults.
- Optional: A dict of the function's keyword defaults `__kwdefaults__`, if the function represented by `__code__` uses any keyword defaults.
- Optional: All other `function attributes <inspect-types>`.

```
class Annotate:
    called_formats = []

    def __call__(self, format=None, /, *, _self=None):
        # When called with fake globals, `_self` will be the
        # actual self value, and `self` will be the format.
        if _self is not None:
            self, format = _self, self

        self.called_formats.append(format)
        if format <= 2:  # VALUE or VALUE_WITH_FAKE_GLOBALS
            return {"x": MyType}
        raise NotImplementedError

    __code__ = __call__.__code__
    __defaults__ = (None,)
    __kwdefaults__ = property(lambda self: dict(_self=self))

    __globals__ = {}
    __builtins__ = {}
    __closure__ = None
```

This can then be called with:

```
>>> from annotationlib import call_annotate_function, Format
>>> call_annotate_function(Annotate(), format=Format.STRING)
{'x': 'MyType'}
```

Or used as the annotate function for an object:

```
>>> from annotationlib import get_annotations, Format
>>> class C:
...   pass
>>> C.__annotate__ = Annotate()
>>> get_annotations(Annotate(), format=Format.STRING)
{'x': 'MyType'}
```

Limitations of the `STRING` format

The `~Format.STRING` format is meant to approximate the source code of the annotation, but the implementation strategy used means that it is not always possible to recover the exact source code.

First, the stringifier of course cannot recover any information that is not present in the compiled code, including comments, whitespace, parenthesization, and operations that get simplified by the compiler.

Second, the stringifier can intercept almost all operations that involve names looked up in some scope, but it cannot intercept operations that operate fully on constants. As a corollary, this also means it is not safe to request the `STRING` format on untrusted code: Python is powerful enough that it is possible to achieve arbitrary code execution even with no access to any globals or builtins. For example:

```
>>> def f(x: (1).__class__.__base__.__subclasses__()[-1].__init__.__builtins__["print"]("Hello
world")): pass
...
>>> annotationlib.get_annotations(f, format=annotationlib.Format.STRING)
Hello world
{'x': 'None'}
```

> [!NOTE] This particular example works as of the time of writing, but it relies on implementation details and is not guaranteed to work in the future.

Among the different kinds of expressions that exist in Python, as represented by the `ast` module, some expressions are supported, meaning that the `STRING` format can generally recover the original source code; others are unsupported, meaning that they may result in incorrect output or an error.

The following are supported (sometimes with caveats):

- `ast.BinOp`
- `ast.UnaryOp`
  - `ast.Invert` (`~`), `ast.UAdd` (`+`), and `ast.USub` (`-`) are supported
  - `ast.Not` (`not`) is not supported
- `ast.Dict` (except when using `**` unpacking)
- `ast.Set`
- `ast.Compare`
  - `ast.Eq` and `ast.NotEq` are supported
  - `ast.Lt`, `ast.LtE`, `ast.Gt`, and `ast.GtE` are supported, but the operand may be flipped
  - `ast.Is`, `ast.IsNot`, `ast.In`, and `ast.NotIn` are not supported
- `ast.Call` (except when using `**` unpacking)
- `ast.Constant` (though not the exact representation of the constant; for example, escape sequences in strings are lost; hexadecimal numbers are converted to decimal)
- `ast.Attribute` (assuming the value is not a constant)
- `ast.Subscript` (assuming the value is not a constant)
- `ast.Starred` (`*` unpacking)
- `ast.Name`
- `ast.List`
- `ast.Tuple`
- `ast.Slice`

The following are unsupported, but throw an informative error when encountered by the stringifier:

- `ast.FormattedValue` (f-strings; error is not detected if conversion specifiers like `!r` are used)
- `ast.JoinedStr` (f-strings)

The following are unsupported and result in incorrect output:

- `ast.BoolOp` (`and` and `or`)
- `ast.IfExp`
- `ast.Lambda`
- `ast.ListComp`
- `ast.SetComp`
- `ast.DictComp`
- `ast.GeneratorExp`

The following are disallowed in annotation scopes and therefore not relevant:

- `ast.NamedExpr` (`:=`)
- `ast.Await`
- `ast.Yield`
- `ast.YieldFrom`

Limitations of the `FORWARDREF` format

The `~Format.FORWARDREF` format aims to produce real values as much as possible, with anything that cannot be resolved replaced with `ForwardRef` objects. It is affected by broadly the same Limitations as the `~Format.STRING` format: annotations that perform operations on literals or that use unsupported expression types may raise exceptions when evaluated using the `~Format.FORWARDREF` format.

Below are a few examples of the behavior with unsupported expressions:

```
>>> from annotationlib import get_annotations, Format
>>> def zerodiv(x: 1 / 0): ...
>>> get_annotations(zerodiv, format=Format.STRING)
Traceback (most recent call last):
  ...
ZeroDivisionError: division by zero
>>> get_annotations(zerodiv, format=Format.FORWARDREF)
Traceback (most recent call last):
  ...
ZeroDivisionError: division by zero
>>> def ifexp(x: 1 if y else 0): ...
>>> get_annotations(ifexp, format=Format.STRING)
{'x': '1'}
```

Security implications of introspecting annotations

Much of the functionality in this module involves executing code related to annotations, which can then do arbitrary things. For example, `get_annotations` may call an arbitrary `annotate function`, and `ForwardRef.evaluate` may call `eval` on an arbitrary string. Code contained in an annotation might make arbitrary system calls, enter an infinite loop, or perform any other operation. This is also true for any access of the `~object.__annotations__` attribute, and for various functions in the `typing` module that work with annotations, such as `typing.get_type_hints`.

Any security issue arising from this also applies immediately after importing code that may contain untrusted annotations: importing code can always cause arbitrary operations to be performed.

However, it is unsafe to accept strings or other input from an untrusted source and pass them to any of the APIs for introspecting annotations, for example by editing an `__annotations__` dictionary or directly creating a `ForwardRef` object.

□□□ #2
□ □□□ □□□ 2026-10-31 17:01:43 UTC
□ □□□ □□□ 2026-10-31 17:01:50 UTC