typing — Support for type hints ¶

Added in version 3.5.

Source code: Lib/typing.py

Note: The Python runtime does not enforce function and variable type annotations. They can be used by third party tools such as type checkers, IDEs, linters, etc.

This module provides runtime support for type hints.

Consider the function below:

```
def surface_area_of_cube(edge_length: float) -> str:
    return f"The surface area of the cube is {6 * edge_length ** 2}."
```

The function `surface_area_of_cube` takes an argument expected to be an instance of `float`, as indicated by the type hint `edge_length: float`. The function is expected to return an instance of `str`, as indicated by the `-> str` hint.

While type hints can be simple classes like `float` or `str`, they can also be more complex. The `typing` module provides a vocabulary of more advanced type hints.

New features are frequently added to the `typing` module. The typing_extensions package provides backports of these new features to older versions of Python.

See also:

"Typing cheat sheet"
A quick overview of type hints (hosted at the mypy docs)

"Type System Reference" section of the mypy docs
The Python typing system is standardised via PEPs, so this reference should broadly apply to most Python type checkers. (Some parts may still be specific to mypy.)

"Static Typing with Python"
Type-checker-agnostic documentation written by the community detailing type system features, useful typing related tools and typing best practices.

Specification for the Python Type System

The canonical, up-to-date specification of the Python type system can be found at "Specification for the Python type system".

Type aliases

```
type Vector = list[float]

def scale(scalar: float, vector: Vector) -> Vector:
    return [scalar * num for num in vector]

# passes type checking; a list of floats qualifies as a Vector.
new_vector = scale(2.0, [1.0, -4.2, 5.4])
```

Type aliases are useful for simplifying complex type signatures. For example:

```
from collections.abc import Sequence

type ConnectionOptions = dict[str, str]
type Address = tuple[str, int]
type Server = tuple[Address, ConnectionOptions]

def broadcast_message(message: str, servers: Sequence[Server]) -> None:
    ...

# The static type checker will treat the previous type signature as
# being exactly equivalent to this one.
def broadcast_message(
    message: str,
    servers: Sequence[tuple[tuple[str, int], dict[str, str]]]
) -> None:
    ...
```

The `type` statement is new in Python 3.12. For backwards compatibility, type aliases can also be created through simple assignment:

```
Vector = list[float]
```

Or marked with `TypeAlias` to make it explicit that this is a type alias, not a normal variable assignment:

```
from typing import TypeAlias

Vector: TypeAlias = list[float]
```

NewType

Use the `NewType` helper to create distinct types:

```
from typing import NewType

UserId = NewType('UserId', int)
some_id = UserId(524313)
```

The static type checker will treat the new type as if it were a subclass of the original type. This is useful in helping catch logical errors:

```
def get_user_name(user_id: UserId) -> str:
    ...

# passes type checking
user_a = get_user_name(UserId(42351))
```

```
user_b = get_user_name(-1)
```

You may still perform all `int` operations on a variable of type `UserId`, but the result will always be of type `int`. This lets you pass in a `UserId` wherever an `int` might be expected, but will prevent you from accidentally creating a `UserId` in an invalid way:

```
# 'output' is of type 'int', not 'UserId'
output = UserId(23413) + UserId(54341)
```

Note that these checks are enforced only by the static type checker. At runtime, the statement `Derived = NewType('Derived', Base)` will make `Derived` a callable that immediately returns whatever parameter you pass it. That means the expression `Derived(some_value)` does not create a new class or introduce much overhead beyond that of a regular function call.

More precisely, the expression `some_value is Derived(some_value)` is always true at runtime.

It is invalid to create a subtype of `Derived`:

```
from typing import NewType

UserId = NewType('UserId', int)

# Fails at runtime and does not pass type checking
class AdminUserId(UserId): pass
```

However, it is possible to create a `NewType` based on a 'derived' `NewType`:

```
from typing import NewType

UserId = NewType('UserId', int)

ProUserId = NewType('ProUserId', UserId)
```

and typechecking for `ProUserId` will work as expected.

See **PEP 484** for more details.

Note: Recall that the use of a type alias declares two types to be *equivalent* to one another. Doing `type Alias = Original` will make the static type checker treat `Alias` as being *exactly equivalent* to `Original` in all cases. This is useful when you want to simplify complex type signatures.

In contrast, `NewType` declares one type to be a *subtype* of another. Doing `Derived = NewType('Derived', Original)` will make the static type checker treat `Derived` as a *subclass* of `Original`, which means a value of type `Original` cannot be used in places where a value of type `Derived` is expected. This is useful when you want to prevent logic errors with minimal runtime cost.

Added in version 3.5.2.

Changed in version 3.10: `NewType` is now a class rather than a function. As a result, there is some additional runtime cost when calling `NewType` over a regular function.

Changed in version 3.11: The performance of calling `NewType` has been restored to its level in Python 3.9.

Functions – or other callable objects – can be annotated using `collections.abc.Callable` or deprecated `typing.Callable`. `Callable[[int], str]` signifies a function that takes a single parameter of type `int` and returns a `str`.

For example:

```
from collections.abc import Callable, Awaitable

def feeder(get_next_item: Callable[[], str]) -> None:
    ...  # Body

def async_query(on_success: Callable[[int], None],
                on_error: Callable[[int, Exception], None]) -> None:
    ...  # Body

async def on_update(value: str) -> None:
    ...  # Body

callback: Callable[[str], Awaitable[None]] = on_update
```

The subscription syntax must always be used with exactly two values: the argument list and the return type. The argument list must be a list of types, a `ParamSpec`, `Concatenate`, or an ellipsis. The return type must be a single type.

If a literal ellipsis `...` is given as the argument list, it indicates that a callable with any arbitrary parameter list would be acceptable:

```
def concat(x: str, y: str) -> str:
    return x + y

x: Callable[..., str]
x = str     # OK
x = concat  # Also OK
```

`Callable` cannot express complex signatures such as functions that take a variadic number of arguments, overloaded functions, or functions that have keyword-only parameters. However, these signatures can be expressed by defining a `Protocol` class with a `__call__()` method:

```
from collections.abc import Iterable
from typing import Protocol

class Combiner(Protocol):
    def __call__(self, *vals: bytes, maxlen: int | None = None) -> list[bytes]: ...

def batch_proc(data: Iterable[bytes], cb_results: Combiner) -> bytes:
    for item in data:
        ...

def good_cb(*vals: bytes, maxlen: int | None = None) -> list[bytes]:
    ...
def bad_cb(*vals: bytes, maxitems: int | None) -> list[bytes]:
    ...

batch_proc([], good_cb)  # OK
```

Callables which take other callables as arguments may indicate that their parameter types are dependent on each other using `ParamSpec`. Additionally, if that callable adds or removes arguments from other callables, the `Concatenate` operator may be used. They take the form `Callable[ParamSpecVariable, ReturnType]` and `Callable[Concatenate[Arg1Type, Arg2Type, ..., ParamSpecVariable], ReturnType]` respectively.

> *Changed in version 3.10:* `Callable` now supports `ParamSpec` and `Concatenate`. See **PEP 612** for more details.

See also: The documentation for `ParamSpec` and `Concatenate` provides examples of usage in `Callable`.

Generics

Since type information about objects kept in containers cannot be statically inferred in a generic way, many container classes in the standard library support subscription to denote the expected types of container elements.

```
from collections.abc import Mapping, Sequence

class Employee: ...

# Sequence[Employee] indicates that all elements in the sequence
# must be instances of "Employee".
# Mapping[str, str] indicates that all keys and all values in the mapping
# must be strings.
def notify_by_email(employees: Sequence[Employee],
                    overrides: Mapping[str, str]) -> None: ...
```

Generic functions and classes can be parameterized by using type parameter syntax:

```
from collections.abc import Sequence

def first[T](l: Sequence[T]) -> T:  # Function is generic over the TypeVar "T"
    return l[0]
```

Or by using the `TypeVar` factory directly:

```
from collections.abc import Sequence
from typing import TypeVar

U = TypeVar('U')                  # Declare type variable "U"

def second(l: Sequence[U]) -> U:  # Function is generic over the TypeVar "U"
    return l[1]
```

> *Changed in version 3.12:* Syntactic support for generics is new in Python 3.12.

Annotating tuples

For most containers in Python, the typing system assumes that all elements in the container will be of the same type. For example:

```
# Type checker will infer that all elements in ``x`` are meant to be ints
x: list[int] = []

# Type checker error: ``list`` only accepts a single type argument:
y: list[int, str] = [1, 'foo']

# Type checker will infer that all keys in ``z`` are meant to be strings,
# and that all values in ``z`` are meant to be either strings or ints
z: Mapping[str, str | int] = {}
```

`list` only accepts one type argument, so a type checker would emit an error on the `y` assignment above. Similarly, `Mapping` only accepts two type arguments: the first indicates the type of the keys, and the second indicates the type of the values.

Unlike most other Python containers, however, it is common in idiomatic Python code for tuples to have elements which are not all of the same type. For this reason, tuples are special-cased in Python's typing system. `tuple` accepts *any number* of type arguments:

```
# OK: ``x`` is assigned to a tuple of length 1 where the sole element is an int
x: tuple[int] = (5,)

# OK: ``y`` is assigned to a tuple of length 2;
# element 1 is an int, element 2 is a str
y: tuple[int, str] = (5, "foo")

# Error: the type annotation indicates a tuple of length 1,
# but ``z`` has been assigned to a tuple of length 3
z: tuple[int] = (1, 2, 3)
```

To denote a tuple which could be of *any* length, and in which all elements are of the same type `T`, use `tuple[T, ...]`. To denote an empty tuple, use `tuple[()]`. Using plain `tuple` as an annotation is equivalent to using `tuple[Any, ...]`:

```
x: tuple[int, ...] = (1, 2)
# These reassignments are OK: ``tuple[int, ...]`` indicates x can be of any length
x = (1, 2, 3)
x = ()
# This reassignment is an error: all elements in ``x`` must be ints
x = ("foo", "bar")

# ``y`` can only ever be assigned to an empty tuple
y: tuple[()] = ()

z: tuple = ("foo", "bar")
# These reassignments are OK: plain ``tuple`` is equivalent to ``tuple[Any, ...]``
z = (1, 2, 3)
z = ()
```

The type of class objects

A variable annotated with `C` may accept a value of type `C`. In contrast, a variable annotated with `type[C]` (or deprecated `typing.Type[C]`) may accept values that are classes themselves – specifically, it will accept the *class object* of `C`. For example:

```
c = type(a)   # Also has type ``type[int]``
```

Note that `type[C]` is covariant:

```
class User: ...
class ProUser(User): ...
class TeamUser(User): ...

def make_new_user(user_class: type[User]) -> User:
    # ...
    return user_class()

make_new_user(User)      # OK
make_new_user(ProUser)   # Also OK: ``type[ProUser]`` is a subtype of ``type[User]``
make_new_user(TeamUser)  # Still fine
make_new_user(User())    # Error: expected ``type[User]`` but got ``User``
make_new_user(int)       # Error: ``type[int]`` is not a subtype of ``type[User]``
```

The only legal parameters for type are classes, Any, type variables, and unions of any of these types. For example:

```
def new_non_team_user(user_class: type[BasicUser | ProUser]): ...

new_non_team_user(BasicUser)  # OK
new_non_team_user(ProUser)    # OK
new_non_team_user(TeamUser)   # Error: ``type[TeamUser]`` is not a subtype
                              # of ``type[BasicUser | ProUser]``
new_non_team_user(User)       # Also an error
```

`type[Any]` is equivalent to type, which is the root of Python's metaclass hierarchy.

Annotating generators and coroutines

A generator can be annotated using the generic type `Generator[YieldType, SendType, ReturnType]`. For example:

```
def echo_round() -> Generator[int, float, str]:
    sent = yield 0
    while sent >= 0:
        sent = yield round(sent)
    return 'Done'
```

Note that unlike many other generic classes in the standard library, the `SendType` of Generator behaves contravariantly, not covariantly or invariantly.

If your generator will only yield values, set the `SendType` and `ReturnType` to `None`:

```
def infinite_stream(start: int) -> Generator[int, None, None]:
    while True:
        yield start
        start += 1
```

Alternatively, annotate your generator as having a return type of either `Iterable[YieldType]` or `Iterator[YieldType]`:

```
        yield start
        start += 1
```

Async generators are handled in a similar fashion, but don't expect a `ReturnType` type argument (`AsyncGenerator[YieldType, SendType]`):

```
async def infinite_stream(start: int) -> AsyncGenerator[int, None]:
    while True:
        yield start
        start = await increment(start)
```

As in the synchronous case, `AsyncIterable[YieldType]` and `AsyncIterator[YieldType]` are available as well:

```
async def infinite_stream(start: int) -> AsyncIterator[int]:
    while True:
        yield start
        start = await increment(start)
```

Coroutines can be annotated using `Coroutine[YieldType, SendType, ReturnType]`. Generic arguments correspond to those of `Generator`, for example:

```
from collections.abc import Coroutine
c: Coroutine[list[str], str, int]  # Some coroutine defined elsewhere
x = c.send('hi')                   # Inferred type of 'x' is list[str]
async def bar() -> None:
    y = await c                    # Inferred type of 'y' is int
```

User-defined generic types

A user-defined class can be defined as a generic class.

```
from logging import Logger

class LoggedVar[T]:
    def __init__(self, value: T, name: str, logger: Logger) -> None:
        self.name = name
        self.logger = logger
        self.value = value

    def set(self, new: T) -> None:
        self.log('Set ' + repr(self.value))
        self.value = new

    def get(self) -> T:
        self.log('Get ' + repr(self.value))
        return self.value

    def log(self, message: str) -> None:
        self.logger.info('%s: %s', self.name, message)
```

This syntax indicates that the class `LoggedVar` is parameterised around a single type variable `T` . This also makes `T` valid as a type within the class body.

```
from typing import TypeVar, Generic

T = TypeVar('T')

class LoggedVar(Generic[T]):
    ...
```

Generic classes have `__class_getitem__()` methods, meaning they can be parameterised at runtime (e.g. `LoggedVar[int]` below):

```
from collections.abc import Iterable

def zero_all_vars(vars: Iterable[LoggedVar[int]]) -> None:
    for var in vars:
        var.set(0)
```

A generic type can have any number of type variables. All varieties of `TypeVar` are permissible as parameters for a generic type:

```
from typing import TypeVar, Generic, Sequence

class WeirdTrio[T, B: Sequence[bytes], S: (int, str)]:
    ...

OldT = TypeVar('OldT', contravariant=True)
OldB = TypeVar('OldB', bound=Sequence[bytes], covariant=True)
OldS = TypeVar('OldS', int, str)

class OldWeirdTrio(Generic[OldT, OldB, OldS]):
    ...
```

Each type variable argument to `Generic` must be distinct. This is thus invalid:

```
from typing import TypeVar, Generic
...

class Pair[M, M]:  # SyntaxError
    ...

T = TypeVar('T')

class Pair(Generic[T, T]):   # INVALID
    ...
```

Generic classes can also inherit from other classes:

```
from collections.abc import Sized

class LinkedList[T](Sized):
    ...
```

When inheriting from generic classes, some type parameters could be fixed:

```
from collections.abc import Mapping
```

In this case `MyDict` has a single parameter, `T`.

Using a generic class without specifying type parameters assumes `Any` for each position. In the following example, `MyIterable` is not generic but implicitly inherits from `Iterable[Any]`:

```
from collections.abc import Iterable

class MyIterable(Iterable): # Same as Iterable[Any]
    ...
```

User-defined generic type aliases are also supported. Examples:

```
from collections.abc import Iterable

type Response[S] = Iterable[S] | int

# Return type here is same as Iterable[str] | int
def response(query: str) -> Response[str]:
    ...

type Vec[T] = Iterable[tuple[T, T]]

def inproduct[T: (int, float, complex)](v: Vec[T]) -> T: # Same as Iterable[tuple[T, T]]
    return sum(x*y for x, y in v)
```

For backward compatibility, generic type aliases can also be created through a simple assignment:

```
from collections.abc import Iterable
from typing import TypeVar

S = TypeVar("S")
Response = Iterable[S] | int
```

> *Changed in version 3.7:* `Generic` no longer has a custom metaclass.

> *Changed in version 3.12:* Syntactic support for generics and type aliases is new in version 3.12. Previously, generic classes had to explicitly inherit from `Generic` or contain a type variable in one of their bases.

User-defined generics for parameter expressions are also supported via parameter specification variables in the form `[**P]`. The behavior is consistent with type variables' described above as parameter specification variables are treated by the typing module as a specialized type variable. The one exception to this is that a list of types can be used to substitute a `ParamSpec`:

```
>>> class Z[T, **P]: ...  # T is a TypeVar; P is a ParamSpec
...
>>> Z[int, [dict, float]]
__main__.Z[int, [dict, float]]
```

Classes generic over a `ParamSpec` can also be created using explicit inheritance from `Generic`. In this case, `**` is not used:

```
from typing import ParamSpec, Generic

P = ParamSpec('P')
```

```
    ...
```

Another difference between `TypeVar` and `ParamSpec` is that a generic with only one parameter specification variable will accept parameter lists in the forms `X[[Type1, Type2, ...]]` and also `X[Type1, Type2, ...]` for aesthetic reasons. Internally, the latter is converted to the former, so the following are equivalent:

```
>>> class X[**P]: ...
...
>>> X[int, str]
__main__.X[[int, str]]
>>> X[[int, str]]
__main__.X[[int, str]]
```

Note that generics with `ParamSpec` may not have correct `__parameters__` after substitution in some cases because they are intended primarily for static type checking.

> *Changed in version 3.10:* `Generic` can now be parameterized over parameter expressions. See `ParamSpec` and **PEP 612** for more details.

A user-defined generic class can have ABCs as base classes without a metaclass conflict. Generic metaclasses are not supported. The outcome of parameterizing generics is cached, and most types in the typing module are hashable and comparable for equality.

The `Any` type

A special kind of type is `Any`. A static type checker will treat every type as being compatible with `Any` and `Any` as being compatible with every type.

This means that it is possible to perform any operation or method call on a value of type `Any` and assign it to any variable:

```
from typing import Any

a: Any = None
a = []          # OK
a = 2           # OK

s: str = ''
s = a           # OK

def foo(item: Any) -> int:
    # Passes type checking; 'item' could be any type,
    # and that type might have a 'bar' method
    item.bar()
    ...
```

Notice that no type checking is performed when assigning a value of type `Any` to a more precise type. For example, the static type checker did not report an error when assigning `a` to `s` even though `s` was declared to be of type `str` and receives an `int` value at runtime!

Furthermore, all functions without a return type or parameter types will implicitly default to using `Any`:

```
def legacy_parser(text):
    ...
```

```
# A static type checker will treat the above
# as having the same signature as:
def legacy_parser(text: Any) -> Any:
    ...
    return data
```

This behavior allows `Any` to be used as an *escape hatch* when you need to mix dynamically and statically typed code.

Contrast the behavior of `Any` with the behavior of `object`. Similar to `Any`, every type is a subtype of `object`. However, unlike `Any`, the reverse is not true: `object` is *not* a subtype of every other type.

That means when the type of a value is `object`, a type checker will reject almost all operations on it, and assigning it to a variable (or using it as a return value) of a more specialized type is a type error. For example:

```
def hash_a(item: object) -> int:
    # Fails type checking; an object does not have a 'magic' method.
    item.magic()
    ...

def hash_b(item: Any) -> int:
    # Passes type checking
    item.magic()
    ...

# Passes type checking, since ints and strs are subclasses of object
hash_a(42)
hash_a("foo")

# Passes type checking, since Any is compatible with all types
hash_b(42)
hash_b("foo")
```

Use `object` to indicate that a value could be any type in a typesafe manner. Use `Any` to indicate that a value is dynamically typed.

Nominal vs structural subtyping

Initially **PEP 484** defined the Python static type system as using *nominal subtyping*. This means that a class `A` is allowed where a class `B` is expected if and only if `A` is a subclass of `B`.

This requirement previously also applied to abstract base classes, such as `Iterable`. The problem with this approach is that a class had to be explicitly marked to support them, which is unpythonic and unlike what one would normally do in idiomatic dynamically typed Python code. For example, this conforms to **PEP 484**:

```
from collections.abc import Sized, Iterable, Iterator

class Bucket(Sized, Iterable[int]):
    ...
    def __len__(self) -> int: ...
    def __iter__(self) -> Iterator[int]: ...
```

PEP 544 allows to solve this problem by allowing users to write the above code without explicit base classes in the class definition, allowing `Bucket` to be implicitly considered a subtype of both `Sized` and `Iterable[int]` by static type checkers. This is known as *structural subtyping* (or static duck-typing):

```
class Bucket:  # Note: no base classes
    ...
    def __len__(self) -> int: ...
    def __iter__(self) -> Iterator[int]: ...

def collect(items: Iterable[int]) -> int: ...
result = collect(Bucket())  # Passes type check
```

Moreover, by subclassing a special class `Protocol`, a user can define new custom protocols to fully enjoy structural subtyping (see examples below).

Module contents

The `typing` module defines the following classes, functions and decorators.

Special typing primitives

Special types

These can be used as types in annotations. They do not support subscription using `[]`.

`typing.`**`Any`**

Special type indicating an unconstrained type.

- Every type is compatible with `Any`.
- `Any` is compatible with every type.

> *Changed in version 3.11:* `Any` can now be used as a base class. This can be useful for avoiding type checker errors with classes that can duck type anywhere or are highly dynamic.

`typing.`**`AnyStr`**

A constrained type variable.

Definition:

```
AnyStr = TypeVar('AnyStr', str, bytes)
```

`AnyStr` is meant to be used for functions that may accept `str` or `bytes` arguments but cannot allow the two to mix.

For example:

```
def concat(a: AnyStr, b: AnyStr) -> AnyStr:
    return a + b

concat("foo", "bar")    # OK, output has type 'str'
concat(b"foo", b"bar")  # OK, output has type 'bytes'
concat("foo", b"bar")   # Error, cannot mix str and bytes
```

Note that, despite its name, `AnyStr` has nothing to do with the `Any` type, nor does it mean “any string”. In particular, `AnyStr` and `str | bytes` are different from each other and have different use cases:

```
# Invalid use of AnyStr:
# The type variable is used only once in the function signature,
```

```
    return "hi there!" if cond else b"greetings!"

# The better way of annotating this function:
def greet_proper(cond: bool) -> str | bytes:
    return "hi there!" if cond else b"greetings!"
```

typing.**LiteralString**

Special type that includes only literal strings.

Any string literal is compatible with `LiteralString`, as is another `LiteralString`. However, an object typed as just `str` is not. A string created by composing `LiteralString`-typed objects is also acceptable as a `LiteralString`.

Example:

```
def run_query(sql: LiteralString) -> None:
    ...

def caller(arbitrary_string: str, literal_string: LiteralString) -> None:
    run_query("SELECT * FROM students")  # OK
    run_query(literal_string)  # OK
    run_query("SELECT * FROM " + literal_string)  # OK
    run_query(arbitrary_string)  # type checker error
    run_query(  # type checker error
        f"SELECT * FROM students WHERE name = {arbitrary_string}"
    )
```

`LiteralString` is useful for sensitive APIs where arbitrary user-generated strings could generate problems. For example, the two cases above that generate type checker errors could be vulnerable to an SQL injection attack.

See **PEP 675** for more details.

> *Added in version 3.11.*

typing.**Never**
typing.**NoReturn**

`Never` and `NoReturn` represent the bottom type, a type that has no members.

They can be used to indicate that a function never returns, such as `sys.exit()`:

```
from typing import Never  # or NoReturn

def stop() -> Never:
    raise RuntimeError('no way')
```

Or to define a function that should never be called, as there are no valid arguments, such as `assert_never()`:

```
from typing import Never  # or NoReturn

def never_call_me(arg: Never) -> None:
    pass

def int_or_str(arg: int | str) -> None:
```

```
        case int():
            print("It's an int")
        case str():
            print("It's a str")
        case _:
            never_call_me(arg)  # OK, arg is of type Never (or NoReturn)
```

`Never` and `NoReturn` have the same meaning in the type system and static type checkers treat both equivalently.

> *Added in version 3.6.2:* Added `NoReturn`.

> *Added in version 3.11:* Added `Never`.

typing.**Self**

Special type to represent the current enclosed class.

For example:

```
from typing import Self, reveal_type

class Foo:
    def return_self(self) -> Self:
        ...
        return self

class SubclassOfFoo(Foo): pass

reveal_type(Foo().return_self())  # Revealed type is "Foo"
reveal_type(SubclassOfFoo().return_self())  # Revealed type is "SubclassOfFoo"
```

This annotation is semantically equivalent to the following, albeit in a more succinct fashion:

```
from typing import TypeVar

Self = TypeVar("Self", bound="Foo")

class Foo:
    def return_self(self: Self) -> Self:
        ...
        return self
```

In general, if something returns `self`, as in the above examples, you should use `Self` as the return annotation. If `Foo.return_self` was annotated as returning `"Foo"`, then the type checker would infer the object returned from `SubclassOfFoo.return_self` as being of type `Foo` rather than `SubclassOfFoo`.

Other common use cases include:

- `classmethod`s that are used as alternative constructors and return instances of the `cls` parameter.
- Annotating an `__enter__()` method which returns self.

You should not use `Self` as the return annotation if the method is not guaranteed to return an instance of a subclass when the class is subclassed:

```
class Eggs:
    # Self would be an incorrect return annotation here,
```

```
    def returns_Eggs(self) -> Eggs:
        return Eggs()
```

See **PEP 673** for more details.

> *Added in version 3.11.*

`typing.`**`TypeAlias`**

Special annotation for explicitly declaring a type alias.

For example:

```
from typing import TypeAlias

Factors: TypeAlias = list[int]
```

`TypeAlias` is particularly useful on older Python versions for annotating aliases that make use of forward references, as it can be hard for type checkers to distinguish these from normal variable assignments:

```
from typing import Generic, TypeAlias, TypeVar

T = TypeVar("T")

# "Box" does not exist yet,
# so we have to use quotes for the forward reference on Python <3.12.
# Using ``TypeAlias`` tells the type checker that this is a type alias declaration,
# not a variable assignment to a string.
BoxOfStrings: TypeAlias = "Box[str]"

class Box(Generic[T]):
    @classmethod
    def make_box_of_strings(cls) -> BoxOfStrings: ...
```

See **PEP 613** for more details.

> *Added in version 3.10.*

> *Deprecated since version 3.12:* `TypeAlias` is deprecated in favor of the `type` statement, which creates instances of `TypeAliasType` and which natively supports forward references. Note that while `TypeAlias` and `TypeAliasType` serve similar purposes and have similar names, they are distinct and the latter is not the type of the former. Removal of `TypeAlias` is not currently planned, but users are encouraged to migrate to `type` statements.

Special forms

These can be used as types in annotations. They all support subscription using `[]`, but each has a unique syntax.

`typing.`**`Union`**

Union type; `Union[X, Y]` is equivalent to `X | Y` and means either X or Y.

To define a union, use e.g. `Union[int, str]` or the shorthand `int | str`. Using that shorthand is recommended. Details:

- Unions of unions are flattened, e.g.:

```
Union[Union[int, str], float] == Union[int, str, float]
```

- Unions of a single argument vanish, e.g.:

```
Union[int] == int  # The constructor actually returns int
```

- Redundant arguments are skipped, e.g.:

```
Union[int, str, int] == Union[int, str] == int | str
```

- When comparing unions, the argument order is ignored, e.g.:

```
Union[int, str] == Union[str, int]
```

- You cannot subclass or instantiate a `Union`.
- You cannot write `Union[X][Y]`.

> *Changed in version 3.7:* Don't remove explicit subclasses from unions at runtime.

> *Changed in version 3.10:* Unions can now be written as `X | Y`. See union type expressions.

typing.**Optional**

`Optional[X]` is equivalent to `X | None` (or `Union[X, None]`).

Note that this is not the same concept as an optional argument, which is one that has a default. An optional argument with a default does not require the `Optional` qualifier on its type annotation just because it is optional. For example:

```
def foo(arg: int = 0) -> None:
    ...
```

On the other hand, if an explicit value of `None` is allowed, the use of `Optional` is appropriate, whether the argument is optional or not. For example:

```
def foo(arg: Optional[int] = None) -> None:
    ...
```

> *Changed in version 3.10:* Optional can now be written as `X | None`. See union type expressions.

typing.**Concatenate**

Special form for annotating higher-order functions.

`Concatenate` can be used in conjunction with Callable and `ParamSpec` to annotate a higher-order callable which adds, removes, or transforms parameters of another callable. Usage is in the form `Concatenate[Arg1Type, Arg2Type, ..., ParamSpecVariable]`. `Concatenate` is currently only valid when used as the first argument to a Callable. The last parameter to `Concatenate` must be a `ParamSpec` or ellipsis (`...`).

the first argument, and returns a callable with a different type signature. In this case, the `ParamSpec` indicates that the returned callable's parameter types are dependent on the parameter types of the callable being passed in:

```
from collections.abc import Callable
from threading import Lock
from typing import Concatenate

# Use this lock to ensure that only one thread is executing a function
# at any time.
my_lock = Lock()

def with_lock[**P, R](f: Callable[Concatenate[Lock, P], R]) -> Callable[P, R]:
    '''A type-safe decorator which provides a lock.'''
    def inner(*args: P.args, **kwargs: P.kwargs) -> R:
        # Provide the lock as the first argument.
        return f(my_lock, *args, **kwargs)
    return inner

@with_lock
def sum_threadsafe(lock: Lock, numbers: list[float]) -> float:
    '''Add a list of numbers together in a thread-safe manner.'''
    with lock:
        return sum(numbers)

# We don't need to pass in the lock ourselves thanks to the decorator.
sum_threadsafe([1.1, 2.2, 3.3])
```

Added in version 3.10.

See also:

- **PEP 612** – Parameter Specification Variables (the PEP which introduced `ParamSpec` and `Concatenate`)
- `ParamSpec`
- Annotating callable objects

typing.**Literal**

Special typing form to define "literal types".

`Literal` can be used to indicate to type checkers that the annotated object has a value equivalent to one of the provided literals.

For example:

```
def validate_simple(data: Any) -> Literal[True]:  # always returns True
    ...

type Mode = Literal['r', 'rb', 'w', 'wb']
def open_helper(file: str, mode: Mode) -> str:
    ...

open_helper('/some/path', 'r')      # Passes type check
open_helper('/other/path', 'typo')  # Error in type checker
```

types.

> *Added in version 3.8.*

> *Changed in version 3.9.1:* `Literal` now de-duplicates parameters. Equality comparisons of `Literal` objects are no longer order dependent. `Literal` objects will now raise a `TypeError` exception during equality comparisons if one of their parameters are not hashable.

typing.ClassVar

Special type construct to mark class variables.

As introduced in **PEP 526**, a variable annotation wrapped in ClassVar indicates that a given attribute is intended to be used as a class variable and should not be set on instances of that class. Usage:

```
class Starship:
    stats: ClassVar[dict[str, int]] = {} # class variable
    damage: int = 10                     # instance variable
```

`ClassVar` accepts only types and cannot be further subscribed.

`ClassVar` is not a class itself, and should not be used with `isinstance()` or `issubclass()`. `ClassVar` does not change Python runtime behavior, but it can be used by third-party type checkers. For example, a type checker might flag the following code as an error:

```
enterprise_d = Starship(3000)
enterprise_d.stats = {} # Error, setting class variable on instance
Starship.stats = {}     # This is OK
```

> *Added in version 3.5.3.*

typing.Final

Special typing construct to indicate final names to type checkers.

Final names cannot be reassigned in any scope. Final names declared in class scopes cannot be overridden in subclasses.

For example:

```
MAX_SIZE: Final = 9000
MAX_SIZE += 1  # Error reported by type checker

class Connection:
    TIMEOUT: Final[int] = 10

class FastConnector(Connection):
    TIMEOUT = 1  # Error reported by type checker
```

There is no runtime checking of these properties. See **PEP 591** for more details.

> *Added in version 3.8.*

typing.Required

Special typing construct to mark a `TypedDict` key as required.

Added in version 3.11.

`typing.`**`NotRequired`**

Special typing construct to mark a `TypedDict` key as potentially missing.

See `TypedDict` and **PEP 655** for more details.

Added in version 3.11.

`typing.`**`Annotated`**

Special typing form to add context-specific metadata to an annotation.

Add metadata `x` to a given type `T` by using the annotation `Annotated[T, x]`. Metadata added using `Annotated` can be used by static analysis tools or at runtime. At runtime, the metadata is stored in a `__metadata__` attribute.

If a library or tool encounters an annotation `Annotated[T, x]` and has no special logic for the metadata, it should ignore the metadata and simply treat the annotation as `T`. As such, `Annotated` can be useful for code that wants to use annotations for purposes outside Python's static typing system.

Using `Annotated[T, x]` as an annotation still allows for static typechecking of `T`, as type checkers will simply ignore the metadata `x`. In this way, `Annotated` differs from the `@no_type_check` decorator, which can also be used for adding annotations outside the scope of the typing system, but completely disables typechecking for a function or class.

The responsibility of how to interpret the metadata lies with the tool or library encountering an `Annotated` annotation. A tool or library encountering an `Annotated` type can scan through the metadata elements to determine if they are of interest (e.g., using `isinstance()`).

`Annotated[<type>, <metadata>]`

Here is an example of how you might use `Annotated` to add metadata to type annotations if you were doing range analysis:

```
@dataclass
class ValueRange:
    lo: int
    hi: int

T1 = Annotated[int, ValueRange(-10, 5)]
T2 = Annotated[T1, ValueRange(-20, 3)]
```

Details of the syntax:

- The first argument to `Annotated` must be a valid type
- Multiple metadata elements can be supplied (`Annotated` supports variadic arguments):

```
@dataclass
class ctype:
    kind: str
```

It is up to the tool consuming the annotations to decide whether the client is allowed to add multiple metadata elements to one annotation and how to merge those annotations.

- `Annotated` must be subscripted with at least two arguments (`Annotated[int]` is not valid)
- The order of the metadata elements is preserved and matters for equality checks:

```
assert Annotated[int, ValueRange(3, 10), ctype("char")] != Annotated[
    int, ctype("char"), ValueRange(3, 10)
]
```

- Nested `Annotated` types are flattened. The order of the metadata elements starts with the innermost annotation:

```
assert Annotated[Annotated[int, ValueRange(3, 10)], ctype("char")] == Annotated[
    int, ValueRange(3, 10), ctype("char")
]
```

- Duplicated metadata elements are not removed:

```
assert Annotated[int, ValueRange(3, 10)] != Annotated[
    int, ValueRange(3, 10), ValueRange(3, 10)
]
```

- `Annotated` can be used with nested and generic aliases:

```
@dataclass
class MaxLen:
    value: int

type Vec[T] = Annotated[list[tuple[T, T]], MaxLen(10)]

# When used in a type annotation, a type checker will treat "V" the same as
# ``Annotated[list[tuple[int, int]], MaxLen(10)]``:
type V = Vec[int]
```

- `Annotated` cannot be used with an unpacked `TypeVarTuple`:

```
type Variadic[*Ts] = Annotated[*Ts, Ann1]  # NOT valid
```

This would be equivalent to:

```
Annotated[T1, T2, T3, ..., Ann1]
```

where `T1`, `T2`, etc. are `TypeVars`. This would be invalid: only one type should be passed to Annotated.

- By default, `get_type_hints()` strips the metadata from annotations. Pass `include_extras=True` to have the metadata preserved:

```
>>> from typing import Annotated, get_type_hints
>>> def func(x: Annotated[int, "metadata"]) -> None: pass
...
>>> get_type_hints(func)
```

{'x': typing.Annotated[int, 'metadata'], 'return': <class 'NoneType'>}

- At runtime, the metadata associated with an `Annotated` type can be retrieved via the `__metadata__` attribute:

```
>>> from typing import Annotated
>>> X = Annotated[int, "very", "important", "metadata"]
>>> X
typing.Annotated[int, 'very', 'important', 'metadata']
>>> X.__metadata__
('very', 'important', 'metadata')
```

- At runtime, if you want to retrieve the original type wrapped by `Annotated`, use the `__origin__` attribute:

```
>>> from typing import Annotated, get_origin
>>> Password = Annotated[str, "secret"]
>>> Password.__origin__
<class 'str'>
```

Note that using `get_origin()` will return `Annotated` itself:

```
>>> get_origin(Password)
<class 'typing.Annotated'>
```

See also:

PEP 593 - Flexible function and variable annotations

The PEP introducing `Annotated` to the standard library.

Added in version 3.9.

typing.TypeGuard

Special typing construct for marking user-defined type guard functions.

`TypeGuard` can be used to annotate the return type of a user-defined type guard function. `TypeGuard` only accepts a single type argument. At runtime, functions marked this way should return a boolean.

`TypeGuard` aims to benefit *type narrowing* – a technique used by static type checkers to determine a more precise type of an expression within a program's code flow. Usually type narrowing is done by analyzing conditional code flow and applying the narrowing to a block of code. The conditional expression here is sometimes referred to as a "type guard":

```
def is_str(val: str | float):
    # "isinstance" type guard
    if isinstance(val, str):
        # Type of ``val`` is narrowed to ``str``
        ...
    else:
        # Else, type of ``val`` is narrowed to ``float``.
        ...
```

Using `-> TypeGuard` tells the static type checker that for a given function:

1. The return value is a boolean.
2. If the return value is `True`, the type of its argument is the type inside `TypeGuard`.

For example:

```
def is_str_list(val: list[object]) -> TypeGuard[list[str]]:
    '''Determines whether all objects in the list are strings'''
    return all(isinstance(x, str) for x in val)

def func1(val: list[object]):
    if is_str_list(val):
        # Type of ``val`` is narrowed to ``list[str]``.
        print(" ".join(val))
    else:
        # Type of ``val`` remains as ``list[object]``.
        print("Not a list of strings!")
```

If `is_str_list` is a class or instance method, then the type in `TypeGuard` maps to the type of the second parameter (after `cls` or `self`).

In short, the form `def foo(arg: TypeA) -> TypeGuard[TypeB]: ...`, means that if `foo(arg)` returns `True`, then `arg` narrows from `TypeA` to `TypeB`.

> **Note:** `TypeB` need not be a narrower form of `TypeA` – it can even be a wider form. The main reason is to allow for things like narrowing `list[object]` to `list[str]` even though the latter is not a subtype of the former, since `list` is invariant. The responsibility of writing type-safe type guards is left to the user.

`TypeGuard` also works with type variables. See **PEP 647** for more details.

> *Added in version 3.10.*

typing.Unpack

Typing operator to conceptually mark an object as having been unpacked.

For example, using the unpack operator `*` on a type variable tuple is equivalent to using `Unpack` to mark the type variable tuple as having been unpacked:

```
Ts = TypeVarTuple('Ts')
tup: tuple[*Ts]
# Effectively does:
tup: tuple[Unpack[Ts]]
```

In fact, `Unpack` can be used interchangeably with `*` in the context of `typing.TypeVarTuple` and `builtins.tuple` types. You might see `Unpack` being used explicitly in older versions of Python, where `*` couldn't be used in certain places:

```
# In older versions of Python, TypeVarTuple and Unpack
# are located in the `typing_extensions` backports package.
```

```
Ts = TypeVarTuple('Ts')
tup: tuple[*Ts]         # Syntax error on Python <= 3.10!
tup: tuple[Unpack[Ts]]  # Semantically equivalent, and backwards-compatible
```

`Unpack` can also be used along with `typing.TypedDict` for typing `**kwargs` in a function signature:

```
from typing import TypedDict, Unpack

class Movie(TypedDict):
    name: str
    year: int

# This function expects two keyword arguments - `name` of type `str`
# and `year` of type `int`.
def foo(**kwargs: Unpack[Movie]): ...
```

See **PEP 692** for more details on using `Unpack` for `**kwargs` typing.

> *Added in version 3.11.*

Building generic types and type aliases

The following classes should not be used directly as annotations. Their intended purpose is to be building blocks for creating generic types and type aliases.

These objects can be created through special syntax (type parameter lists and the `type` statement). For compatibility with Python 3.11 and earlier, they can also be created without the dedicated syntax, as documented below.

class typing.**Generic**

Abstract base class for generic types.

A generic type is typically declared by adding a list of type parameters after the class name:

```
class Mapping[KT, VT]:
    def __getitem__(self, key: KT) -> VT:
        ...
        # Etc.
```

Such a class implicitly inherits from `Generic`. The runtime semantics of this syntax are discussed in the Language Reference.

This class can then be used as follows:

```
def lookup_name[X, Y](mapping: Mapping[X, Y], key: X, default: Y) -> Y:
    try:
        return mapping[key]
    except KeyError:
        return default
```

Here the brackets after the function name indicate a generic function.

For backwards compatibility, generic classes can also be declared by explicitly inheriting from `Generic`. In this case, the type parameters must be declared separately:

```
class Mapping(Generic[KT, VT]):
    def __getitem__(self, key: KT) -> VT:
        ...
        # Etc.
```

class typing.**TypeVar**(*name*, **constraints*, *bound=None*, *covariant=False*, *contravariant=False*, *infer_variance=False*)

Type variable.

The preferred way to construct a type variable is via the dedicated syntax for generic functions, generic classes, and generic type aliases:

```
class Sequence[T]:  # T is a TypeVar
    ...
```

This syntax can also be used to create bound and constrained type variables:

```
class StrSequence[S: str]:  # S is a TypeVar bound to str
    ...


class StrOrBytesSequence[A: (str, bytes)]:  # A is a TypeVar constrained to str or bytes
    ...
```

However, if desired, reusable type variables can also be constructed manually, like so:

```
T = TypeVar('T')  # Can be anything
S = TypeVar('S', bound=str)  # Can be any subtype of str
A = TypeVar('A', str, bytes)  # Must be exactly str or bytes
```

Type variables exist primarily for the benefit of static type checkers. They serve as the parameters for generic types as well as for generic function and type alias definitions. See `Generic` for more information on generic types. Generic functions work as follows:

```
def repeat[T](x: T, n: int) -> Sequence[T]:
    """Return a list containing n references to x."""
    return [x]*n


def print_capitalized[S: str](x: S) -> S:
    """Print x capitalized, and return x."""
    print(x.capitalize())
    return x


def concatenate[A: (str, bytes)](x: A, y: A) -> A:
    """Add two strings or bytes objects together."""
    return x + y
```

Note that type variables can be *bound*, *constrained*, or neither, but cannot be both bound *and* constrained.

The variance of type variables is inferred by type checkers when they are created through the type parameter syntax or when `infer_variance=True` is passed. Manually created type variables may be explicitly

Bound type variables and constrained type variables have different semantics in several important ways. Using a *bound* type variable means that the `TypeVar` will be solved using the most specific type possible:

```
x = print_capitalized('a string')
reveal_type(x)  # revealed type is str

class StringSubclass(str):
    pass

y = print_capitalized(StringSubclass('another string'))
reveal_type(y)  # revealed type is StringSubclass

z = print_capitalized(45)  # error: int is not a subtype of str
```

Type variables can be bound to concrete types, abstract types (ABCs or protocols), and even unions of types:

```
# Can be anything with an __abs__ method
def print_abs[T: SupportsAbs](arg: T) -> None:
    print("Absolute value:", abs(arg))

U = TypeVar('U', bound=str|bytes)  # Can be any subtype of the union str|bytes
V = TypeVar('V', bound=SupportsAbs)  # Can be anything with an __abs__ method
```

Using a *constrained* type variable, however, means that the `TypeVar` can only ever be solved as being exactly one of the constraints given:

```
a = concatenate('one', 'two')
reveal_type(a)  # revealed type is str

b = concatenate(StringSubclass('one'), StringSubclass('two'))
reveal_type(b)  # revealed type is str, despite StringSubclass being passed in

c = concatenate('one', b'two')  # error: type variable 'A' can be either str or bytes ir
```

At runtime, `isinstance(x, T)` will raise `TypeError`.

`__name__`

The name of the type variable.

`__covariant__`

Whether the type var has been explicitly marked as covariant.

`__contravariant__`

Whether the type var has been explicitly marked as contravariant.

`__infer_variance__`

Whether the type variable's variance should be inferred by type checkers.

> *Added in version 3.12.*

The bound of the type variable, if any.

> *Changed in version 3.12:* For type variables created through type parameter syntax, the bound is evaluated only when the attribute is accessed, not when the type variable is created (see Lazy evaluation).

__constraints__

A tuple containing the constraints of the type variable, if any.

> *Changed in version 3.12:* For type variables created through type parameter syntax, the constraints are evaluated only when the attribute is accessed, not when the type variable is created (see Lazy evaluation).

> *Changed in version 3.12:* Type variables can now be declared using the type parameter syntax introduced by **PEP 695**. The `infer_variance` parameter was added.

class typing.**TypeVarTuple**(*name*)

Type variable tuple. A specialized form of type variable that enables *variadic* generics.

Type variable tuples can be declared in type parameter lists using a single asterisk (`*`) before the name:

```
def move_first_element_to_last[T, *Ts](tup: tuple[T, *Ts]) -> tuple[*Ts, T]:
    return (*tup[1:], tup[0])
```

Or by explicitly invoking the `TypeVarTuple` constructor:

```
T = TypeVar("T")
Ts = TypeVarTuple("Ts")

def move_first_element_to_last(tup: tuple[T, *Ts]) -> tuple[*Ts, T]:
    return (*tup[1:], tup[0])
```

A normal type variable enables parameterization with a single type. A type variable tuple, in contrast, allows parameterization with an *arbitrary* number of types by acting like an *arbitrary* number of type variables wrapped in a tuple. For example:

```
# T is bound to int, Ts is bound to ()
# Return value is (1,), which has type tuple[int]
move_first_element_to_last(tup=(1,))

# T is bound to int, Ts is bound to (str,)
# Return value is ('spam', 1), which has type tuple[str, int]
move_first_element_to_last(tup=(1, 'spam'))

# T is bound to int, Ts is bound to (str, float)
# Return value is ('spam', 3.0, 1), which has type tuple[str, float, int]
move_first_element_to_last(tup=(1, 'spam', 3.0))

# This fails to type check (and fails at runtime)
# because tuple[()] is not compatible with tuple[T, *Ts]
# (at least one element is required)
move_first_element_to_last(tup=())
```

which is equivalent to `tuple[T, T1, T2, ...]`. (Note that in older versions of Python, you might see this written using `Unpack` instead, as `Unpack[Ts]`.)

Type variable tuples must *always* be unpacked. This helps distinguish type variable tuples from normal type variables:

```
x: Ts          # Not valid
x: tuple[Ts]   # Not valid
x: tuple[*Ts]  # The correct way to do it
```

Type variable tuples can be used in the same contexts as normal type variables. For example, in class definitions, arguments, and return types:

```
class Array[*Shape]:
    def __getitem__(self, key: tuple[*Shape]) -> float: ...
    def __abs__(self) -> "Array[*Shape]": ...
    def get_shape(self) -> tuple[*Shape]: ...
```

Type variable tuples can be happily combined with normal type variables:

```
class Array[DType, *Shape]:  # This is fine
    pass

class Array2[*Shape, DType]:  # This would also be fine
    pass

class Height: ...
class Width: ...

float_array_1d: Array[float, Height] = Array()     # Totally fine
int_array_2d: Array[int, Height, Width] = Array()  # Yup, fine too
```

However, note that at most one type variable tuple may appear in a single list of type arguments or type parameters:

```
x: tuple[*Ts, *Ts]            # Not valid
class Array[*Shape, *Shape]:  # Not valid
    pass
```

Finally, an unpacked type variable tuple can be used as the type annotation of `*args`:

```
def call_soon[*Ts](
    callback: Callable[[*Ts], None],
    *args: *Ts
) -> None:
    ...
    callback(*args)
```

In contrast to non-unpacked annotations of `*args` - e.g. `*args: int`, which would specify that *all* arguments are `int` - `*args: *Ts` enables reference to the types of the *individual* arguments in `*args`. Here, this allows us to ensure the types of the `*args` passed to `call_soon` match the types of the (positional) arguments of `callback`.

__name__

The name of the type variable tuple.

Added in version 3.11.

Changed in version 3.12: Type variable tuples can now be declared using the type parameter syntax introduced by **PEP 695**.

class typing.**ParamSpec**(*name*, *, *bound=None*, *covariant=False*, *contravariant=False*)

Parameter specification variable. A specialized version of type variables.

In type parameter lists, parameter specifications can be declared with two asterisks (**):

```
type IntFunc[**P] = Callable[P, int]
```

For compatibility with Python 3.11 and earlier, ParamSpec objects can also be created as follows:

```
P = ParamSpec('P')
```

Parameter specification variables exist primarily for the benefit of static type checkers. They are used to forward the parameter types of one callable to another callable – a pattern commonly found in higher order functions and decorators. They are only valid when used in Concatenate, or as the first argument to Callable, or as parameters for user-defined Generics. See Generic for more information on generic types.

For example, to add basic logging to a function, one can create a decorator add_logging to log function calls. The parameter specification variable tells the type checker that the callable passed into the decorator and the new callable returned by it have inter-dependent type parameters:

```
from collections.abc import Callable
import logging

def add_logging[T, **P](f: Callable[P, T]) -> Callable[P, T]:
    '''A type-safe decorator to add logging to a function.'''
    def inner(*args: P.args, **kwargs: P.kwargs) -> T:
        logging.info(f'{f.__name__} was called')
        return f(*args, **kwargs)
    return inner

@add_logging
def add_two(x: float, y: float) -> float:
    '''Add two numbers together.'''
    return x + y
```

Without ParamSpec, the simplest way to annotate this previously was to use a TypeVar with bound Callable[..., Any]. However this causes two problems:

1. The type checker can't type check the inner function because *args and **kwargs have to be typed Any.
2. cast() may be required in the body of the add_logging decorator when returning the inner function, or the static type checker must be told to ignore the return inner.

kwargs

Since `ParamSpec` captures both positional and keyword parameters, `P.args` and `P.kwargs` can be used to split a `ParamSpec` into its components. `P.args` represents the tuple of positional parameters in a given call and should only be used to annotate `*args`. `P.kwargs` represents the mapping of keyword parameters to their values in a given call, and should be only be used to annotate `**kwargs`. Both attributes require the annotated parameter to be in scope. At runtime, `P.args` and `P.kwargs` are instances respectively of `ParamSpecArgs` and `ParamSpecKwargs`.

__name__

The name of the parameter specification.

Parameter specification variables created with `covariant=True` or `contravariant=True` can be used to declare covariant or contravariant generic types. The `bound` argument is also accepted, similar to `TypeVar`. However the actual semantics of these keywords are yet to be decided.

> *Added in version 3.10.*

> *Changed in version 3.12:* Parameter specifications can now be declared using the type parameter syntax introduced by **PEP 695**.

Note: Only parameter specification variables defined in global scope can be pickled.

See also:

- **PEP 612** – Parameter Specification Variables (the PEP which introduced `ParamSpec` and `Concatenate`)
- `Concatenate`
- Annotating callable objects

typing.**ParamSpecArgs**
typing.**ParamSpecKwargs**

Arguments and keyword arguments attributes of a `ParamSpec`. The `P.args` attribute of a `ParamSpec` is an instance of `ParamSpecArgs`, and `P.kwargs` is an instance of `ParamSpecKwargs`. They are intended for runtime introspection and have no special meaning to static type checkers.

Calling `get_origin()` on either of these objects will return the original `ParamSpec`:

```
>>> from typing import ParamSpec, get_origin
>>> P = ParamSpec("P")
>>> get_origin(P.args) is P
True
>>> get_origin(P.kwargs) is P
True
```

> *Added in version 3.10.*

class typing.**TypeAliasType**(*name*, *value*, *, *type_params=()*)

The type of type aliases created through the `type` statement.

```
>>> type Alias = int
>>> type(Alias)
<class 'typing.TypeAliasType'>
```

Added in version 3.12.

__name__

The name of the type alias:

```
>>> type Alias = int
>>> Alias.__name__
'Alias'
```

__module__

The module in which the type alias was defined:

```
>>> type Alias = int
>>> Alias.__module__
'__main__'
```

__type_params__

The type parameters of the type alias, or an empty tuple if the alias is not generic:

```
>>> type ListOrSet[T] = list[T] | set[T]
>>> ListOrSet.__type_params__
(T,)
>>> type NotGeneric = int
>>> NotGeneric.__type_params__
()
```

__value__

The type alias's value. This is lazily evaluated, so names used in the definition of the alias are not resolved until the `__value__` attribute is accessed:

```
>>> type Mutually = Recursive
>>> type Recursive = Mutually
>>> Mutually
Mutually
>>> Recursive
Recursive
>>> Mutually.__value__
Recursive
>>> Recursive.__value__
Mutually
```

Other special directives

These functions and classes should not be used directly as annotations. Their intended purpose is to be building blocks for creating and declaring types.

class typing.**NamedTuple**

Typed version of `collections.namedtuple()`.

```
class Employee(NamedTuple):
    name: str
    id: int
```

This is equivalent to:

```
Employee = collections.namedtuple('Employee', ['name', 'id'])
```

To give a field a default value, you can assign to it in the class body:

```
class Employee(NamedTuple):
    name: str
    id: int = 3

employee = Employee('Guido')
assert employee.id == 3
```

Fields with a default value must come after any fields without a default.

The resulting class has an extra attribute `__annotations__` giving a dict that maps the field names to the field types. (The field names are in the `_fields` attribute and the default values are in the `_field_defaults` attribute, both of which are part of the `namedtuple()` API.)

`NamedTuple` subclasses can also have docstrings and methods:

```
class Employee(NamedTuple):
    """Represents an employee."""
    name: str
    id: int = 3

    def __repr__(self) -> str:
        return f'<Employee {self.name}, id={self.id}>'
```

`NamedTuple` subclasses can be generic:

```
class Group[T](NamedTuple):
    key: T
    group: list[T]
```

Backward-compatible usage:

```
# For creating a generic NamedTuple on Python 3.11 or lower
class Group(NamedTuple, Generic[T]):
    key: T
    group: list[T]

# A functional syntax is also supported
Employee = NamedTuple('Employee', [('name', str), ('id', int)])
```

> *Changed in version 3.6:* Added support for **PEP 526** variable annotation syntax.

> *Changed in version 3.6.1:* Added support for default values, methods, and docstrings.

> *Changed in version 3.9:* Removed the `_field_types` attribute in favor of the more standard `__annotations__` attribute which has the same information.

> *Changed in version 3.11:* Added support for generic namedtuples.

class typing.**NewType**(*name*, *tp*)

Helper class to create low-overhead distinct types.

A `NewType` is considered a distinct type by a typechecker. At runtime, however, calling a `NewType` returns its argument unchanged.

Usage:

```
UserId = NewType('UserId', int)  # Declare the NewType "UserId"
first_user = UserId(1)  # "UserId" returns the argument unchanged at runtime
```

__module__

The module in which the new type is defined.

__name__

The name of the new type.

__supertype__

The type that the new type is based on.

> *Added in version 3.5.2.*

> *Changed in version 3.10:* `NewType` is now a class rather than a function.

class typing.**Protocol**(*Generic*)

Base class for protocol classes.

Protocol classes are defined like this:

```
class Proto(Protocol):
    def meth(self) -> int:
        ...
```

Such classes are primarily used with static type checkers that recognize structural subtyping (static duck-typing), for example:

```
class C:
    def meth(self) -> int:
        return 0

def func(x: Proto) -> int:
    return x.meth()

func(C())  # Passes static type check
```

signatures.

Protocol classes can be generic, for example:

```
class GenProto[T](Protocol):
    def meth(self) -> T:
        ...
```

In code that needs to be compatible with Python 3.11 or older, generic Protocols can be written as follows:

```
T = TypeVar("T")

class GenProto(Protocol[T]):
    def meth(self) -> T:
        ...
```

Added in version 3.8.

@typing.**runtime_checkable**

Mark a protocol class as a runtime protocol.

Such a protocol can be used with isinstance() and issubclass(). This raises TypeError when applied to a non-protocol class. This allows a simple-minded structural check, very similar to "one trick ponies" in collections.abc such as Iterable. For example:

```
@runtime_checkable
class Closable(Protocol):
    def close(self): ...

assert isinstance(open('/some/file'), Closable)

@runtime_checkable
class Named(Protocol):
    name: str

import threading
assert isinstance(threading.Thread(name='Bob'), Named)
```

Note: runtime_checkable() will check only the presence of the required methods or attributes, not their type signatures or types. For example, ssl.SSLObject is a class, therefore it passes an issubclass() check against Callable. However, the ssl.SSLObject.__init__ method exists only to raise a TypeError with a more informative message, therefore making it impossible to call (instantiate) ssl.SSLObject.

Note: An isinstance() check against a runtime-checkable protocol can be surprisingly slow compared to an isinstance() check against a non-protocol class. Consider using alternative idioms such as hasattr() calls for structural checks in performance-sensitive code.

Added in version 3.8.

hasattr() was used). As a result, some objects which used to be considered instances of a runtime-checkable protocol may no longer be considered instances of that protocol on Python 3.12+, and vice versa. Most users are unlikely to be affected by this change.

Changed in version 3.12: The members of a runtime-checkable protocol are now considered "frozen" at runtime as soon as the class has been created. Monkey-patching attributes onto a runtime-checkable protocol will still work, but will have no impact on isinstance() checks comparing objects to the protocol. See "What's new in Python 3.12" for more details.

class typing.**TypedDict**(*dict*)

Special construct to add type hints to a dictionary. At runtime it is a plain dict.

TypedDict declares a dictionary type that expects all of its instances to have a certain set of keys, where each key is associated with a value of a consistent type. This expectation is not checked at runtime but is only enforced by type checkers. Usage:

```
class Point2D(TypedDict):
    x: int
    y: int
    label: str

a: Point2D = {'x': 1, 'y': 2, 'label': 'good'}  # OK
b: Point2D = {'z': 3, 'label': 'bad'}           # Fails type check

assert Point2D(x=1, y=2, label='first') == dict(x=1, y=2, label='first')
```

To allow using this feature with older versions of Python that do not support **PEP 526**, TypedDict supports two additional equivalent syntactic forms:

- Using a literal dict as the second argument:

```
Point2D = TypedDict('Point2D', {'x': int, 'y': int, 'label': str})
```

- Using keyword arguments:

```
Point2D = TypedDict('Point2D', x=int, y=int, label=str)
```

Deprecated since version 3.11, will be removed in version 3.13: The keyword-argument syntax is deprecated in 3.11 and will be removed in 3.13. It may also be unsupported by static type checkers.

The functional syntax should also be used when any of the keys are not valid identifiers, for example because they are keywords or contain hyphens. Example:

```
# raises SyntaxError
class Point2D(TypedDict):
    in: int  # 'in' is a keyword
    x-y: int  # name with hyphens

# OK, functional syntax
Point2D = TypedDict('Point2D', {'in': int, 'x-y': int})
```

```
class Point2D(TypedDict):
    x: int
    y: int
    label: NotRequired[str]

# Alternative syntax
Point2D = TypedDict('Point2D', {'x': int, 'y': int, 'label': NotRequired[str]})
```

This means that a `Point2D TypedDict` can have the `label` key omitted.

It is also possible to mark all keys as non-required by default by specifying a totality of `False`:

```
class Point2D(TypedDict, total=False):
    x: int
    y: int

# Alternative syntax
Point2D = TypedDict('Point2D', {'x': int, 'y': int}, total=False)
```

This means that a `Point2D TypedDict` can have any of the keys omitted. A type checker is only expected to support a literal `False` or `True` as the value of the `total` argument. `True` is the default, and makes all items defined in the class body required.

Individual keys of a `total=False TypedDict` can be marked as required using `Required`:

```
class Point2D(TypedDict, total=False):
    x: Required[int]
    y: Required[int]
    label: str

# Alternative syntax
Point2D = TypedDict('Point2D', {
    'x': Required[int],
    'y': Required[int],
    'label': str
}, total=False)
```

It is possible for a `TypedDict` type to inherit from one or more other `TypedDict` types using the class-based syntax. Usage:

```
class Point3D(Point2D):
    z: int
```

`Point3D` has three items: `x`, `y` and `z`. It is equivalent to this definition:

```
class Point3D(TypedDict):
    x: int
    y: int
    z: int
```

A `TypedDict` cannot inherit from a non-`TypedDict` class, except for `Generic`. For example:

```
class X(TypedDict):
    x: int
```

```
    y: int

class Z(object): pass  # A non-TypedDict class

class XY(X, Y): pass  # OK

class XZ(X, Z): pass  # raises TypeError
```

A `TypedDict` can be generic:

```
class Group[T](TypedDict):
    key: T
    group: list[T]
```

To create a generic `TypedDict` that is compatible with Python 3.11 or lower, inherit from `Generic` explicitly:

```
T = TypeVar("T")

class Group(TypedDict, Generic[T]):
    key: T
    group: list[T]
```

A `TypedDict` can be introspected via annotations dicts (see Annotations Best Practices for more information on annotations best practices), `__total__`, `__required_keys__`, and `__optional_keys__`.

`__total__`

`Point2D.__total__` gives the value of the `total` argument. Example:

```
>>> from typing import TypedDict
>>> class Point2D(TypedDict): pass
>>> Point2D.__total__
True
>>> class Point2D(TypedDict, total=False): pass
>>> Point2D.__total__
False
>>> class Point3D(Point2D): pass
>>> Point3D.__total__
True
```

This attribute reflects *only* the value of the `total` argument to the current `TypedDict` class, not whether the class is semantically total. For example, a `TypedDict` with `__total__` set to `True` may have keys marked with `NotRequired`, or it may inherit from another `TypedDict` with `total=False`. Therefore, it is generally better to use `__required_keys__` and `__optional_keys__` for introspection.

`__required_keys__`

> *Added in version 3.9.*

`__optional_keys__`

`Point2D.__required_keys__` and `Point2D.__optional_keys__` return `frozenset` objects containing required and non-required keys, respectively.

For backwards compatibility with Python 3.10 and below, it is also possible to use inheritance to declare both required and non-required keys in the same `TypedDict` . This is done by declaring a `TypedDict` with one value for the `total` argument and then inheriting from it in another `TypedDict` with a different value for `total`:

```
>>> class Point2D(TypedDict, total=False):
...     x: int
...     y: int
...
>>> class Point3D(Point2D):
...     z: int
...
>>> Point3D.__required_keys__ == frozenset({'z'})
True
>>> Point3D.__optional_keys__ == frozenset({'x', 'y'})
True
```

Added in version 3.9.

Note: If `from __future__ import annotations` is used or if annotations are given as strings, annotations are not evaluated when the `TypedDict` is defined. Therefore, the runtime introspection that `__required_keys__` and `__optional_keys__` rely on may not work properly, and the values of the attributes may be incorrect.

See **PEP 589** for more examples and detailed rules of using `TypedDict`.

Added in version 3.8.

Changed in version 3.11: Added support for marking individual keys as `Required` or `NotRequired`. See **PEP 655**.

Changed in version 3.11: Added support for generic `TypedDicts`.

Protocols

The following protocols are provided by the typing module. All are decorated with `@runtime_checkable`.

class typing.**SupportsAbs**
An ABC with one abstract method `__abs__` that is covariant in its return type.

class typing.**SupportsBytes**
An ABC with one abstract method `__bytes__`.

class typing.**SupportsComplex**
An ABC with one abstract method `__complex__`.

class typing.**SupportsFloat**
An ABC with one abstract method `__float__`.

class typing.**SupportsIndex**
An ABC with one abstract method `__index__`.

class typing.**SupportsInt**

An ABC with one abstract method `__int__`.

class typing.**SupportsRound**

An ABC with one abstract method `__round__` that is covariant in its return type.

ABCs for working with IO

class typing.**IO**
class typing.**TextIO**
class typing.**BinaryIO**

Generic type `IO[AnyStr]` and its subclasses `TextIO(IO[str])` and `BinaryIO(IO[bytes])` represent the types of I/O streams such as returned by `open()`.

Functions and decorators

typing.**cast**(*typ*, *val*)

Cast a value to a type.

This returns the value unchanged. To the type checker this signals that the return value has the designated type, but at runtime we intentionally don't check anything (we want this to be as fast as possible).

typing.**assert_type**(*val*, *typ*, */*)

Ask a static type checker to confirm that *val* has an inferred type of *typ*.

At runtime this does nothing: it returns the first argument unchanged with no checks or side effects, no matter the actual type of the argument.

When a static type checker encounters a call to `assert_type()`, it emits an error if the value is not of the specified type:

```
def greet(name: str) -> None:
    assert_type(name, str)  # OK, inferred type of `name` is `str`
    assert_type(name, int)  # type checker error
```

This function is useful for ensuring the type checker's understanding of a script is in line with the developer's intentions:

```
def complex_function(arg: object):
    # Do some complex type-narrowing logic,
    # after which we hope the inferred type will be `int`
    ...
    # Test whether the type checker correctly understands our function
    assert_type(arg, int)
```

Added in version 3.11.

typing.**assert_never**(*arg*, */*)

Ask a static type checker to confirm that a line of code is unreachable.

Example:

```
        case int():
            print("It's an int")
        case str():
            print("It's a str")
        case _ as unreachable:
            assert_never(unreachable)
```

Here, the annotations allow the type checker to infer that the last case can never execute, because `arg` is either an `int` or a `str`, and both options are covered by earlier cases.

If a type checker finds that a call to `assert_never()` is reachable, it will emit an error. For example, if the type annotation for `arg` was instead `int | str | float`, the type checker would emit an error pointing out that `unreachable` is of type `float`. For a call to `assert_never` to pass type checking, the inferred type of the argument passed in must be the bottom type, `Never`, and nothing else.

At runtime, this throws an exception when called.

See also: Unreachable Code and Exhaustiveness Checking has more information about exhaustiveness checking with static typing.

Added in version 3.11.

`typing.`**`reveal_type`**`(`*`obj`*`, /)`

Ask a static type checker to reveal the inferred type of an expression.

When a static type checker encounters a call to this function, it emits a diagnostic with the inferred type of the argument. For example:

```
x: int = 1
reveal_type(x)  # Revealed type is "builtins.int"
```

This can be useful when you want to debug how your type checker handles a particular piece of code.

At runtime, this function prints the runtime type of its argument to `sys.stderr` and returns the argument unchanged (allowing the call to be used within an expression):

```
x = reveal_type(1)  # prints "Runtime type is int"
print(x)  # prints "1"
```

Note that the runtime type may be different from (more or less specific than) the type statically inferred by a type checker.

Most type checkers support `reveal_type()` anywhere, even if the name is not imported from `typing`. Importing the name from `typing`, however, allows your code to run without runtime errors and communicates intent more clearly.

Added in version 3.11.

`@typing.`**`dataclass_transform`**`(`*`*, eq_default=True, order_default=False, kw_only_default=False, frozen_default=False, field_specifiers=(), **kwargs`*`)`

Decorator to mark an object as providing `dataclass`-like behavior.

runtime “magic” that transforms a class in a similar way to `@dataclasses.dataclass`.

Example usage with a decorator function:

```
@dataclass_transform()
def create_model[T](cls: type[T]) -> type[T]:
    ...
    return cls

@create_model
class CustomerModel:
    id: int
    name: str
```

On a base class:

```
@dataclass_transform()
class ModelBase: ...

class CustomerModel(ModelBase):
    id: int
    name: str
```

On a metaclass:

```
@dataclass_transform()
class ModelMeta(type): ...

class ModelBase(metaclass=ModelMeta): ...

class CustomerModel(ModelBase):
    id: int
    name: str
```

The `CustomerModel` classes defined above will be treated by type checkers similarly to classes created with `@dataclasses.dataclass`. For example, type checkers will assume these classes have `__init__` methods that accept `id` and `name`.

The decorated class, metaclass, or function may accept the following bool arguments which type checkers will assume have the same effect as they would have on the `@dataclasses.dataclass` decorator: `init`, `eq`, `order`, `unsafe_hash`, `frozen`, `match_args`, `kw_only`, and `slots`. It must be possible for the value of these arguments (`True` or `False`) to be statically evaluated.

The arguments to the `dataclass_transform` decorator can be used to customize the default behaviors of the decorated class, metaclass, or function:

Parameters:

- **eq_default** (*bool*) – Indicates whether the `eq` parameter is assumed to be `True` or `False` if it is omitted by the caller. Defaults to `True`.
- **order_default** (*bool*) – Indicates whether the `order` parameter is assumed to be `True` or `False` if it is omitted by the caller. Defaults to `False`.
- **kw_only_default** (*bool*) – Indicates whether the `kw_only` parameter is assumed to be `True` or `False` if it is omitted by the caller. Defaults to `False`.
- **frozen_default** (*bool*) –

Added in version 3.12.

- **field_specifiers** (*tuple*[*Callable*[..., *Any*], ...]) – Specifies a static list of supported classes or functions that describe fields, similar to `dataclasses.field()`. Defaults to `()`.
- ****kwargs** (*Any*) – Arbitrary other keyword arguments are accepted in order to allow for possible future extensions.

Type checkers recognize the following optional parameters on field specifiers:

Recognised parameters for field specifiers

Parameter name	Description
`init`	Indicates whether the field should be included in the synthesized `__init__` method. If unspecified, `init` defaults to `True`.
`default`	Provides the default value for the field.
`default_factory`	Provides a runtime callback that returns the default value for the field. If neither `default` nor `default_factory` are specified, the field is assumed to have no default value and must be provided a value when the class is instantiated.
`factory`	An alias for the `default_factory` parameter on field specifiers.
`kw_only`	Indicates whether the field should be marked as keyword-only. If `True`, the field will be keyword-only. If `False`, it will not be keyword-only. If unspecified, the value of the `kw_only` parameter on the object decorated with `dataclass_transform` will be used, or if that is unspecified, the value of `kw_only_default` on `dataclass_transform` will be used.
`alias`	Provides an alternative name for the field. This alternative name is used in the synthesized `__init__` method.

At runtime, this decorator records its arguments in the `__dataclass_transform__` attribute on the decorated object. It has no other runtime effect.

See **PEP 681** for more details.

Added in version 3.11.

`@typing.overload`

Decorator for creating overloaded functions and methods.

The `@overload` decorator allows describing functions and methods that support multiple different combinations of argument types. A series of `@overload`-decorated definitions must be followed by exactly one non-`@overload`-decorated definition (for the same function/method).

`@overload`-decorated definitions are for the benefit of the type checker only, since they will be overwritten by the non-`@overload`-decorated definition. The non-`@overload`-decorated definition, meanwhile, will be used at runtime but should be ignored by a type checker. At runtime, calling an `@overload`-decorated function directly will raise `NotImplementedError`.

```
@overload
def process(response: None) -> None:
    ...
@overload
def process(response: int) -> tuple[int, str]:
    ...
@overload
def process(response: bytes) -> str:
    ...
def process(response):
    ...  # actual implementation goes here
```

See **PEP 484** for more details and comparison with other typing semantics.

> *Changed in version 3.11:* Overloaded functions can now be introspected at runtime using `get_overloads()`.

`typing.`**`get_overloads`**`(`*`func`*`)`

Return a sequence of `@overload`-decorated definitions for *func*.

func is the function object for the implementation of the overloaded function. For example, given the definition of `process` in the documentation for `@overload`, `get_overloads(process)` will return a sequence of three function objects for the three defined overloads. If called on a function with no overloads, `get_overloads()` returns an empty sequence.

`get_overloads()` can be used for introspecting an overloaded function at runtime.

> *Added in version 3.11.*

`typing.`**`clear_overloads`**`()`

Clear all registered overloads in the internal registry.

This can be used to reclaim the memory used by the registry.

> *Added in version 3.11.*

`@typing.`**`final`**

Decorator to indicate final methods and final classes.

Decorating a method with `@final` indicates to a type checker that the method cannot be overridden in a subclass. Decorating a class with `@final` indicates that it cannot be subclassed.

For example:

```
class Base:
    @final
    def done(self) -> None:
        ...
class Sub(Base):
    def done(self) -> None:  # Error reported by type checker
        ...

@final
```

```
class Other(Leaf):  # Error reported by type checker
    ...
```

There is no runtime checking of these properties. See **PEP 591** for more details.

> *Added in version 3.8.*

> *Changed in version 3.11:* The decorator will now attempt to set a `__final__` attribute to `True` on the decorated object. Thus, a check like `if getattr(obj, "__final__", False)` can be used at runtime to determine whether an object `obj` has been marked as final. If the decorated object does not support setting attributes, the decorator returns the object unchanged without raising an exception.

`@typing.no_type_check`

Decorator to indicate that annotations are not type hints.

This works as a class or function decorator. With a class, it applies recursively to all methods and classes defined in that class (but not to methods defined in its superclasses or subclasses). Type checkers will ignore all annotations in a function or class with this decorator.

`@no_type_check` mutates the decorated object in place.

`@typing.no_type_check_decorator`

Decorator to give another decorator the `no_type_check()` effect.

This wraps the decorator with something that wraps the decorated function in `no_type_check()`.

`@typing.override`

Decorator to indicate that a method in a subclass is intended to override a method or attribute in a superclass.

Type checkers should emit an error if a method decorated with `@override` does not, in fact, override anything. This helps prevent bugs that may occur when a base class is changed without an equivalent change to a child class.

For example:

```
class Base:
    def log_status(self) -> None:
        ...

class Sub(Base):
    @override
    def log_status(self) -> None:  # Okay: overrides Base.log_status
        ...

    @override
    def done(self) -> None:  # Error reported by type checker
        ...
```

There is no runtime checking of this property.

The decorator will attempt to set an `__override__` attribute to `True` on the decorated object. Thus, a check like `if getattr(obj, "__override__", False)` can be used at runtime to determine whether an

See **PEP 698** for more details.

> *Added in version 3.12.*

`@typing.type_check_only`

Decorator to mark a class or function as unavailable at runtime.

This decorator is itself not available at runtime. It is mainly intended to mark classes that are defined in type stub files if an implementation returns an instance of a private class:

```
@type_check_only
class Response:  # private or not available at runtime
    code: int
    def get_header(self, name: str) -> str: ...

def fetch_response() -> Response: ...
```

Note that returning instances of private classes is not recommended. It is usually preferable to make such classes public.

Introspection helpers

`typing.get_type_hints(obj, globalns=None, localns=None, include_extras=False)`

Return a dictionary containing type hints for a function, method, module or class object.

This is often the same as `obj.__annotations__`, but this function makes the following changes to the annotations dictionary:

- Forward references encoded as string literals or `ForwardRef` objects are handled by evaluating them in *globalns*, *localns*, and (where applicable) *obj*'s type parameter namespace. If *globalns* or *localns* is not given, appropriate namespace dictionaries are inferred from *obj*.
- `None` is replaced with `types.NoneType`.
- If `@no_type_check` has been applied to *obj*, an empty dictionary is returned.
- If *obj* is a class `C`, the function returns a dictionary that merges annotations from `C`'s base classes with those on `C` directly. This is done by traversing `C.__mro__` and iteratively combining `__annotations__` dictionaries. Annotations on classes appearing earlier in the method resolution order always take precedence over annotations on classes appearing later in the method resolution order.
- The function recursively replaces all occurrences of `Annotated[T, ...]` with `T`, unless *include_extras* is set to `True` (see `Annotated` for more information).

See also `inspect.get_annotations()`, a lower-level function that returns annotations more directly.

> **Note:** If any forward references in the annotations of *obj* are not resolvable or are not valid Python code, this function will raise an exception such as `NameError`. For example, this can happen with imported type aliases that include forward references, or with names imported under `if TYPE_CHECKING`.

> *Changed in version 3.9:* Added `include_extras` parameter as part of **PEP 593**. See the documentation on `Annotated` for more information.

typing.**get_origin**(*tp*)

Get the unsubscripted version of a type: for a typing object of the form `X[Y, Z, ...]` return `X`.

If `X` is a typing-module alias for a builtin or `collections` class, it will be normalized to the original class. If `X` is an instance of `ParamSpecArgs` or `ParamSpecKwargs`, return the underlying `ParamSpec`. Return `None` for unsupported objects.

Examples:

```
assert get_origin(str) is None
assert get_origin(Dict[str, int]) is dict
assert get_origin(Union[int, str]) is Union
assert get_origin(Annotated[str, "metadata"]) is Annotated
P = ParamSpec('P')
assert get_origin(P.args) is P
assert get_origin(P.kwargs) is P
```

Added in version 3.8.

typing.**get_args**(*tp*)

Get type arguments with all substitutions performed: for a typing object of the form `X[Y, Z, ...]` return `(Y, Z, ...)`.

If `X` is a union or `Literal` contained in another generic type, the order of `(Y, Z, ...)` may be different from the order of the original arguments `[Y, Z, ...]` due to type caching. Return `()` for unsupported objects.

Examples:

```
assert get_args(int) == ()
assert get_args(Dict[int, str]) == (int, str)
assert get_args(Union[int, str]) == (int, str)
```

Added in version 3.8.

typing.**is_typeddict**(*tp*)

Check if a type is a `TypedDict`.

For example:

```
class Film(TypedDict):
    title: str
    year: int

assert is_typeddict(Film)
assert not is_typeddict(list | str)

# TypedDict is a factory for creating typed dicts,
# not a typed dict itself
assert not is_typeddict(TypedDict)
```

Added in version 3.10.

For example, `List["SomeClass"]` is implicitly transformed into `List[ForwardRef("SomeClass")]`. `ForwardRef` should not be instantiated by a user, but may be used by introspection tools.

Note: PEP 585 generic types such as `list["SomeClass"]` will not be implicitly transformed into `list[ForwardRef("SomeClass")]` and thus will not automatically resolve to `list[SomeClass]`.

Added in version 3.7.4.

Constant

`typing.`**`TYPE_CHECKING`**

A special constant that is assumed to be `True` by 3rd party static type checkers. It is `False` at runtime.

Usage:

```
if TYPE_CHECKING:
    import expensive_mod

def fun(arg: 'expensive_mod.SomeType') -> None:
    local_var: expensive_mod.AnotherType = other_fun()
```

The first type annotation must be enclosed in quotes, making it a "forward reference", to hide the `expensive_mod` reference from the interpreter runtime. Type annotations for local variables are not evaluated, so the second annotation does not need to be enclosed in quotes.

Note: If `from __future__ import annotations` is used, annotations are not evaluated at function definition time. Instead, they are stored as strings in `__annotations__`. This makes it unnecessary to use quotes around the annotation (see PEP 563).

Added in version 3.5.2.

Deprecated aliases

This module defines several deprecated aliases to pre-existing standard library classes. These were originally included in the typing module in order to support parameterizing these generic classes using `[]`. However, the aliases became redundant in Python 3.9 when the corresponding pre-existing classes were enhanced to support `[]` (see PEP 585).

The redundant types are deprecated as of Python 3.9. However, while the aliases may be removed at some point, removal of these aliases is not currently planned. As such, no deprecation warnings are currently issued by the interpreter for these aliases.

If at some point it is decided to remove these deprecated aliases, a deprecation warning will be issued by the interpreter for at least two releases prior to removal. The aliases are guaranteed to remain in the typing module without deprecation warnings until at least Python 3.14.

Type checkers are encouraged to flag uses of the deprecated types if the program they are checking targets a minimum Python version of 3.9 or newer.

class typing.**Dict**(*dict, MutableMapping[KT, VT]*)

Deprecated alias to `dict`.

Note that to annotate arguments, it is preferred to use an abstract collection type such as `Mapping` rather than to use `dict` or `typing.Dict`.

> *Deprecated since version 3.9:* `builtins.dict` now supports subscripting (`[]`). See **PEP 585** and Generic Alias Type.

class typing.**List**(*list, MutableSequence[T]*)

Deprecated alias to `list`.

Note that to annotate arguments, it is preferred to use an abstract collection type such as `Sequence` or `Iterable` rather than to use `list` or `typing.List`.

> *Deprecated since version 3.9:* `builtins.list` now supports subscripting (`[]`). See **PEP 585** and Generic Alias Type.

class typing.**Set**(*set, MutableSet[T]*)

Deprecated alias to `builtins.set`.

Note that to annotate arguments, it is preferred to use an abstract collection type such as `collections.abc.Set` rather than to use `set` or `typing.Set`.

> *Deprecated since version 3.9:* `builtins.set` now supports subscripting (`[]`). See **PEP 585** and Generic Alias Type.

class typing.**FrozenSet**(*frozenset, AbstractSet[T_co]*)

Deprecated alias to `builtins.frozenset`.

> *Deprecated since version 3.9:* `builtins.frozenset` now supports subscripting (`[]`). See **PEP 585** and Generic Alias Type.

typing.**Tuple**

Deprecated alias for `tuple`.

`tuple` and `Tuple` are special-cased in the type system; see Annotating tuples for more details.

> *Deprecated since version 3.9:* `builtins.tuple` now supports subscripting (`[]`). See **PEP 585** and Generic Alias Type.

class typing.**Type**(*Generic[CT_co]*)

Deprecated alias to `type`.

See The type of class objects for details on using `type` or `typing.Type` in type annotations.

> *Added in version 3.5.2.*

> *Deprecated since version 3.9:* `builtins.type` now supports subscripting (`[]`). See **PEP 585** and Generic Alias Type.

class typing.**DefaultDict**(*collections.defaultdict, MutableMapping[KT, VT]*)

Deprecated alias to `collections.defaultdict`.

Added in version 3.5.2.

Deprecated since version 3.9: `collections.defaultdict` now supports subscripting (`[]`). See **PEP 585** and Generic Alias Type.

class typing.**OrderedDict**(*collections.OrderedDict, MutableMapping[KT, VT]*)

Deprecated alias to `collections.OrderedDict`.

Added in version 3.7.2.

Deprecated since version 3.9: `collections.OrderedDict` now supports subscripting (`[]`). See **PEP 585** and Generic Alias Type.

class typing.**ChainMap**(*collections.ChainMap, MutableMapping[KT, VT]*)

Deprecated alias to `collections.ChainMap`.

Added in version 3.6.1.

Deprecated since version 3.9: `collections.ChainMap` now supports subscripting (`[]`). See **PEP 585** and Generic Alias Type.

class typing.**Counter**(*collections.Counter, Dict[T, int]*)

Deprecated alias to `collections.Counter`.

Added in version 3.6.1.

Deprecated since version 3.9: `collections.Counter` now supports subscripting (`[]`). See **PEP 585** and Generic Alias Type.

class typing.**Deque**(*deque, MutableSequence[T]*)

Deprecated alias to `collections.deque`.

Added in version 3.6.1.

Deprecated since version 3.9: `collections.deque` now supports subscripting (`[]`). See **PEP 585** and Generic Alias Type.

Aliases to other concrete types

Deprecated since version 3.8, will be removed in version 3.13: The `typing.io` namespace is deprecated and will be removed. These types should be directly imported from `typing` instead.

class typing.**Pattern**
class typing.**Match**

Deprecated aliases corresponding to the return types from `re.compile()` and `re.match()`.

These types (and the corresponding functions) are generic over `AnyStr`. `Pattern` can be specialised as `Pattern[str]` or `Pattern[bytes]`; `Match` can be specialised as `Match[str]` or `Match[bytes]`.

> *Deprecated since version 3.9:* Classes `Pattern` and `Match` from `re` now support `[]`. See **PEP 585** and Generic Alias Type.

class typing.**Text**

Deprecated alias for `str`.

`Text` is provided to supply a forward compatible path for Python 2 code: in Python 2, `Text` is an alias for `unicode`.

Use `Text` to indicate that a value must contain a unicode string in a manner that is compatible with both Python 2 and Python 3:

```
def add_unicode_checkmark(text: Text) -> Text:
    return text + u' \u2713'
```

> *Added in version 3.5.2.*

> *Deprecated since version 3.11:* Python 2 is no longer supported, and most type checkers also no longer support type checking Python 2 code. Removal of the alias is not currently planned, but users are encouraged to use `str` instead of `Text`.

Aliases to container ABCs in `collections.abc`

class typing.**AbstractSet**(*Collection[T_co]*)

Deprecated alias to `collections.abc.Set`.

> *Deprecated since version 3.9:* `collections.abc.Set` now supports subscripting (`[]`). See **PEP 585** and Generic Alias Type.

class typing.**ByteString**(*Sequence[int]*)

This type represents the types `bytes`, `bytearray`, and `memoryview` of byte sequences.

> *Deprecated since version 3.9, will be removed in version 3.14:* Prefer `collections.abc.Buffer`, or a union like `bytes | bytearray | memoryview`.

class typing.**Collection**(*Sized, Iterable[T_co], Container[T_co]*)

Deprecated alias to `collections.abc.Collection`.

> *Added in version 3.6.*

> *Deprecated since version 3.9:* `collections.abc.Collection` now supports subscripting (`[]`). See **PEP 585** and Generic Alias Type.

class typing.**Container**(*Generic[T_co]*)

Deprecated alias to `collections.abc.Container`.

> *Deprecated since version 3.9:* `collections.abc.Container` now supports subscripting (`[]`). See **PEP 585** and Generic Alias Type.

class typing.**ItemsView**(*MappingView, AbstractSet[tuple[KT_co, VT_co]]*)

> *Deprecated since version 3.9:* `collections.abc.ItemsView` now supports subscripting (`[]`). See **PEP 585** and Generic Alias Type.

class typing.**KeysView**(*MappingView, AbstractSet[KT_co]*)

Deprecated alias to `collections.abc.KeysView`.

> *Deprecated since version 3.9:* `collections.abc.KeysView` now supports subscripting (`[]`). See **PEP 585** and Generic Alias Type.

class typing.**Mapping**(*Collection[KT], Generic[KT, VT_co]*)

Deprecated alias to `collections.abc.Mapping`.

> *Deprecated since version 3.9:* `collections.abc.Mapping` now supports subscripting (`[]`). See **PEP 585** and Generic Alias Type.

class typing.**MappingView**(*Sized*)

Deprecated alias to `collections.abc.MappingView`.

> *Deprecated since version 3.9:* `collections.abc.MappingView` now supports subscripting (`[]`). See **PEP 585** and Generic Alias Type.

class typing.**MutableMapping**(*Mapping[KT, VT]*)

Deprecated alias to `collections.abc.MutableMapping`.

> *Deprecated since version 3.9:* `collections.abc.MutableMapping` now supports subscripting (`[]`). See **PEP 585** and Generic Alias Type.

class typing.**MutableSequence**(*Sequence[T]*)

Deprecated alias to `collections.abc.MutableSequence`.

> *Deprecated since version 3.9:* `collections.abc.MutableSequence` now supports subscripting (`[]`). See **PEP 585** and Generic Alias Type.

class typing.**MutableSet**(*AbstractSet[T]*)

Deprecated alias to `collections.abc.MutableSet`.

> *Deprecated since version 3.9:* `collections.abc.MutableSet` now supports subscripting (`[]`). See **PEP 585** and Generic Alias Type.

class typing.**Sequence**(*Reversible[T_co], Collection[T_co]*)

Deprecated alias to `collections.abc.Sequence`.

> *Deprecated since version 3.9:* `collections.abc.Sequence` now supports subscripting (`[]`). See **PEP 585** and Generic Alias Type.

class typing.**ValuesView**(*MappingView, Collection[_VT_co]*)

Deprecated alias to `collections.abc.ValuesView`.

> *Deprecated since version 3.9:* `collections.abc.ValuesView` now supports subscripting (`[]`). See **PEP 585** and Generic Alias Type.

class typing.**Coroutine**(*Awaitable[ReturnType], Generic[YieldType, SendType, ReturnType]*)

Deprecated alias to `collections.abc.Coroutine`.

See Annotating generators and coroutines for details on using `collections.abc.Coroutine` and `typing.Coroutine` in type annotations.

> *Added in version 3.5.3.*

> *Deprecated since version 3.9:* `collections.abc.Coroutine` now supports subscripting (`[]`). See **PEP 585** and Generic Alias Type.

class typing.**AsyncGenerator**(*AsyncIterator[YieldType], Generic[YieldType, SendType]*)

Deprecated alias to `collections.abc.AsyncGenerator`.

See Annotating generators and coroutines for details on using `collections.abc.AsyncGenerator` and `typing.AsyncGenerator` in type annotations.

> *Added in version 3.6.1.*

> *Deprecated since version 3.9:* `collections.abc.AsyncGenerator` now supports subscripting (`[]`). See **PEP 585** and Generic Alias Type.

class typing.**AsyncIterable**(*Generic[T_co]*)

Deprecated alias to `collections.abc.AsyncIterable`.

> *Added in version 3.5.2.*

> *Deprecated since version 3.9:* `collections.abc.AsyncIterable` now supports subscripting (`[]`). See **PEP 585** and Generic Alias Type.

class typing.**AsyncIterator**(*AsyncIterable[T_co]*)

Deprecated alias to `collections.abc.AsyncIterator`.

> *Added in version 3.5.2.*

> *Deprecated since version 3.9:* `collections.abc.AsyncIterator` now supports subscripting (`[]`). See **PEP 585** and Generic Alias Type.

class typing.**Awaitable**(*Generic[T_co]*)

Deprecated alias to `collections.abc.Awaitable`.

> *Added in version 3.5.2.*

> *Deprecated since version 3.9:* `collections.abc.Awaitable` now supports subscripting (`[]`). See **PEP 585** and Generic Alias Type.

Aliases to other ABCs in `collections.abc`

class typing.**Iterable**(*Generic[T_co]*)

Deprecated alias to `collections.abc.Iterable`.

class typing.**Iterator**(*Iterable[T_co]*)

Deprecated alias to collections.abc.Iterator.

> *Deprecated since version 3.9:* collections.abc.Iterator now supports subscripting ([]). See **PEP 585** and Generic Alias Type.

typing.**Callable**

Deprecated alias to collections.abc.Callable.

See Annotating callable objects for details on how to use collections.abc.Callable and typing.Callable in type annotations.

> *Deprecated since version 3.9:* collections.abc.Callable now supports subscripting ([]). See **PEP 585** and Generic Alias Type.

> *Changed in version 3.10:* Callable now supports ParamSpec and Concatenate. See **PEP 612** for more details.

class typing.**Generator**(*Iterator[YieldType], Generic[YieldType, SendType, ReturnType]*)

Deprecated alias to collections.abc.Generator.

See Annotating generators and coroutines for details on using collections.abc.Generator and typing.Generator in type annotations.

> *Deprecated since version 3.9:* collections.abc.Generator now supports subscripting ([]). See **PEP 585** and Generic Alias Type.

class typing.**Hashable**

Deprecated alias to collections.abc.Hashable.

> *Deprecated since version 3.12:* Use collections.abc.Hashable directly instead.

class typing.**Reversible**(*Iterable[T_co]*)

Deprecated alias to collections.abc.Reversible.

> *Deprecated since version 3.9:* collections.abc.Reversible now supports subscripting ([]). See **PEP 585** and Generic Alias Type.

class typing.**Sized**

Deprecated alias to collections.abc.Sized.

> *Deprecated since version 3.12:* Use collections.abc.Sized directly instead.

Aliases to contextlib ABCs

class typing.**ContextManager**(*Generic[T_co]*)

Deprecated alias to contextlib.AbstractContextManager.

> *Added in version 3.5.4.*

class typing.**AsyncContextManager**(*Generic[T_co]*)

Deprecated alias to `contextlib.AbstractAsyncContextManager`.

> *Added in version 3.6.2.*

> *Deprecated since version 3.9:* `contextlib.AbstractAsyncContextManager` now supports subscripting (`[]`). See **PEP 585** and Generic Alias Type.

Deprecation Timeline of Major Features

Certain features in `typing` are deprecated and may be removed in a future version of Python. The following table summarizes major deprecations for your convenience. This is subject to change, and not all deprecations are listed.

Feature	Deprecated in	Projected removal	PEP/issue
`typing.io` and `typing.re` submodules	3.8	3.13	bpo-38291
`typing` versions of standard collections	3.9	Undecided (see Deprecated aliases for more information)	**PEP 585**
`typing.ByteString`	3.9	3.14	gh-91896
`typing.Text`	3.11	Undecided	gh-92332
`typing.Hashable` and `typing.Sized`	3.12	Undecided	gh-94309
`typing.TypeAlias`	3.12	Undecided	**PEP 695**