26.1. typing — Support for type hints

New in version 3.5.

Source code: Lib/typing.py

> **Note:** The typing module has been included in the standard library on a provisional basis. New features might be added and API may change even between minor releases if deemed necessary by the core developers.

This module supports type hints as specified by **PEP 484** and **PEP 526**. The most fundamental support consists of the types `Any`, `Union`, `Tuple`, `Callable`, `TypeVar`, and `Generic`. For full specification please see **PEP 484**. For a simplified introduction to type hints see **PEP 483**.

The function below takes and returns a string and is annotated as follows:

```
def greeting(name: str) -> str:
    return 'Hello ' + name
```

In the function `greeting`, the argument `name` is expected to be of type `str` and the return type `str`. Subtypes are accepted as arguments.

26.1.1. Type aliases

A type alias is defined by assigning the type to the alias. In this example, `Vector` and `List[float]` will be treated as interchangeable synonyms:

```
from typing import List
Vector = List[float]

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

# typechecks; 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 typing import Dict, Tuple, List

ConnectionOptions = Dict[str, str]
Address = Tuple[str, int]
Server = Tuple[Address, ConnectionOptions]
```

```
def broadcast_message(message: str, servers: List[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: List[Tuple[Tuple[str, int], Dict[str, str]]]) -> None
    ...
```

Note that `None` as a type hint is a special case and is replaced by `type(None)`.

26.1.2. NewType

Use the `NewType()` helper function 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:
    ...

# typechecks
user_a = get_user_name(UserId(42351))

# does not typecheck; an int is not a UserId
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 function that immediately returns whatever parameter you pass it. That means the expression `Derived(some_value)` does not create a new class or introduce any overhead beyond that of a regular function call.

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

This also means that it is not possible to create a subtype of `Derived` since it is an identity function at runtime, not an actual type:

```
from typing import NewType

UserId = NewType('UserId', int)

# Fails at runtime and does not typecheck
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 `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.

New in version 3.5.2.

26.1.3. Callable

Frameworks expecting callback functions of specific signatures might be type hinted using `Callable[[Arg1Type, Arg2Type], ReturnType]`.

For example:

```
from typing import Callable

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
```

It is possible to declare the return type of a callable without specifying the call signature by substituting a literal ellipsis for the list of arguments in the type hint: `Callable[..., ReturnType]`.

26.1.4. Generics

Since type information about objects kept in containers cannot be statically inferred in a generic way, abstract base classes have been extended to support subscription to denote expected types for container elements.

```
from typing import Mapping, Sequence

def notify_by_email(employees: Sequence[Employee],
                    overrides: Mapping[str, str]) -> None: ...
```

Generics can be parametrized by using a new factory available in typing called `TypeVar`.

```
from typing import Sequence, TypeVar

T = TypeVar('T')      # Declare type variable

def first(l: Sequence[T]) -> T:   # Generic function
    return l[0]
```

26.1.5. User-defined generic types

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

```
from typing import TypeVar, Generic
from logging import Logger

T = TypeVar('T')

class LoggedVar(Generic[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)
```

`Generic[T]` as a base class defines that the class `LoggedVar` takes a single type parameter `T` . This also makes `T` valid as a type within the class body.

The `Generic` base class uses a metaclass that defines `__getitem__()` so that `LoggedVar[t]` is valid as a type:

```
from typing 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, and type variables may be constrained:

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

T = TypeVar('T')
S = TypeVar('S', int, str)

class StrangePair(Generic[T, S]):
    ...
```

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

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

T = TypeVar('T')

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

You can use multiple inheritance with `Generic`:

```python
from typing import TypeVar, Generic, Sized

T = TypeVar('T')

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

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

```python
from typing import TypeVar, Mapping

T = TypeVar('T')

class MyDict(Mapping[str, T]):
    ...
```

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]`:

```python
from typing import Iterable

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

User defined generic type aliases are also supported. Examples:

```python
from typing import TypeVar, Iterable, Tuple, Union
S = TypeVar('S')
Response = Union[Iterable[S], int]

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

T = TypeVar('T', int, float, complex)
Vec = Iterable[Tuple[T, T]]

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

The metaclass used by `Generic` is a subclass of `abc.ABCMeta`. A generic class can be an ABC by including abstract methods or properties, and generic classes can also 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.

26.1.6. 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 on `Any` and assign it to any variable:

```
from typing import Any

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

s = ''      # type: str
s = a       # OK

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

Notice that no typechecking 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):
    ...
    return data

# 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; an object does not have a 'magic' method.
    item.magic()
    ...

def hash_b(item: Any) -> int:
    # Typechecks
    item.magic()
    ...

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

# Typechecks, 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.

26.1.7. Classes, functions, and decorators

The module defines the following classes, functions and decorators:

class `typing.`**`TypeVar`**

Type variable.

Usage:

```
T = TypeVar('T')  # Can be anything
A = TypeVar('A', str, bytes)  # Must be 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 definitions. See class Generic for more information on generic types. Generic functions work as follows:

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

def longest(x: A, y: A) -> A:
```

```
    """Return the longest of two strings."""
    return x if len(x) >= len(y) else y
```

The latter example's signature is essentially the overloading of `(str, str) -> str` and `(bytes, bytes) -> bytes`. Also note that if the arguments are instances of some subclass of `str`, the return type is still plain `str`.

At runtime, `isinstance(x, T)` will raise `TypeError`. In general, `isinstance()` and `issubclass()` should not be used with types.

Type variables may be marked covariant or contravariant by passing `covariant=True` or `contravariant=True`. See **PEP 484** for more details. By default type variables are invariant. Alternatively, a type variable may specify an upper bound using `bound=<type>`. This means that an actual type substituted (explicitly or implicitly) for the type variable must be a subclass of the boundary type, see **PEP 484**.

class `typing.`**`Generic`**

Abstract base class for generic types.

A generic type is typically declared by inheriting from an instantiation of this class with one or more type variables. For example, a generic mapping type might be defined as:

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

This class can then be used as follows:

```
X = TypeVar('X')
Y = TypeVar('Y')

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

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

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

```
a = 3         # Has type 'int'
b = int       # Has type 'Type[int]'
c = type(a)   # Also has type 'Type[int]'
```

Note that `Type[C]` is covariant:

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

# Accepts User, BasicUser, ProUser, TeamUser, ...
def make_new_user(user_class: Type[User]) -> User:
    # ...
    return user_class()
```

The fact that `Type[C]` is covariant implies that all subclasses of `C` should implement the same constructor signature and class method signatures as `C`. The type checker should flag violations of this, but should also allow constructor calls in subclasses that match the constructor calls in the indicated base class. How the type checker is required to handle this particular case may change in future revisions of **PEP 484**.

The only legal parameters for `Type` are classes, unions of classes, and `Any`. For example:

```
def new_non_team_user(user_class: Type[Union[BaseUser, ProUser]]):
```

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

New in version 3.5.2.

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

A generic version of `collections.abc.Iterable`.

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

A generic version of `collections.abc.Iterator`.

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

A generic version of `collections.abc.Reversible`.

class `typing.`**`SupportsInt`**

An ABC with one abstract method `__int__`.

class `typing.`**`SupportsFloat`**

An ABC with one abstract method `__float__`.

class `typing.`**`SupportsComplex`**

An ABC with one abstract method `__complex__`.

class `typing.`**`SupportsBytes`**

An ABC with one abstract method `__bytes__`.

class `typing.`**`SupportsAbs`**

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

class `typing.`**`SupportsRound`**

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

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

A generic version of `collections.abc.Container`.

class `typing.`**`Hashable`**

An alias to `collections.abc.Hashable`

class `typing.`**`Sized`**

An alias to `collections.abc.Sized`

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

A generic version of `collections.abc.Collection`

New in version 3.6.

class `typing.`**`AbstractSet`**(*Sized, Collection[T_co]*)

A generic version of `collections.abc.Set`.

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

A generic version of `collections.abc.MutableSet`.

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

A generic version of `collections.abc.Mapping`.

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

A generic version of `collections.abc.MutableMapping`.

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

A generic version of `collections.abc.Sequence`.

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

A generic version of `collections.abc.MutableSequence`.

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

A generic version of `collections.abc.ByteString`.

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

As a shorthand for this type, `bytes` can be used to annotate arguments of any of the types mentioned above.

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

A generic version of `collections.deque`.

New in version 3.6.1.

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

Generic version of `list`. Useful for annotating return types. To annotate arguments it is preferred to use abstract collection types such as `Mapping`, `Sequence`, or `AbstractSet`.

This type may be used as follows:

```
T = TypeVar('T', int, float)

def vec2(x: T, y: T) -> List[T]:
    return [x, y]

def keep_positives(vector: Sequence[T]) -> List[T]:
    return [item for item in vector if item > 0]
```

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

A generic version of `builtins.set`.

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

A generic version of `builtins.frozenset`.

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

A generic version of `collections.abc.MappingView`.

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

A generic version of `collections.abc.KeysView`.

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

A generic version of `collections.abc.ItemsView`.

class typing.**ValuesView**(*MappingView[VT_co]*)

A generic version of `collections.abc.ValuesView`.

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

A generic version of `collections.abc.Awaitable`.

class `typing.`**`Coroutine`**(*Awaitable[V_co], Generic[T_co T_contra, V_co]*)

A generic version of `collections.abc.Coroutine`. The variance and order of type variables correspond to those of `Generator`, for example:

```
from typing import List, Coroutine
c = None # type: Coroutine[List[str], str, int]
...
x = c.send('hi') # type: List[str]
async def bar() -> None:
    x = await c # type: int
```

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

A generic version of `collections.abc.AsyncIterable`.

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

A generic version of `collections.abc.AsyncIterator`.

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

A generic version of `contextlib.AbstractContextManager`.

New in version 3.6.

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

An ABC with async abstract `__aenter__()` and `__aexit__()` methods.

New in version 3.6.

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

A generic version of `dict`. The usage of this type is as follows:

```
def get_position_in_index(word_list: Dict[str, int], word: str) ->
    return word_list[word]
```

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

A generic version of `collections.defaultdict`.

New in version 3.5.2.

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

A generic version of `collections.Counter`.

New in version 3.6.1.

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

A generic version of `collections.ChainMap`.

New in version 3.6.1.

class `typing.`**`Generator`**(*Iterator[T_co], Generic[T_co, T_contra, V_co]*)

A generator can be annotated by 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 generics in the typing module, 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]`:

```
def infinite_stream(start: int) -> Iterator[int]:
    while True:
        yield start
        start += 1
```

class `typing.`**`AsyncGenerator`**(*AsyncIterator[T_co], Generic[T_co, T_contra]*)

An async generator can be annotated by the generic type `AsyncGenerator[YieldType, SendType]`. For example:

```
async def echo_round() -> AsyncGenerator[int, float]:
    sent = yield 0
    while sent >= 0.0:
        rounded = await round(sent)
        sent = yield rounded
```

Unlike normal generators, async generators cannot return a value, so there is no `ReturnType` type parameter. As with `Generator`, the `SendType` behaves contravariantly.

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

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

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

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

New in version 3.5.4.

class `typing.`**`Text`**

`Text` is an alias for `str`. It 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'
```

New in version 3.5.2.

class `typing.`**`io`**

Wrapper namespace for I/O stream types.

This defines the generic type `IO[AnyStr]` and subclasses `TextIO` and `BinaryIO`, deriving from `IO[str]` and `IO[bytes]`, respectively. These represent the types of I/O streams such as returned by `open()`.

These types are also accessible directly as `typing.IO`, `typing.TextIO`, and `typing.BinaryIO`.

class `typing.`**`re`**

Wrapper namespace for regular expression matching types.

This defines the type aliases `Pattern` and `Match` which correspond to the return types from `re.compile()` and `re.match()`. These types (and the corresponding functions) are generic in `AnyStr` and can be made specific by writing `Pattern[str]`, `Pattern[bytes]`, `Match[str]`, or `Match[bytes]`.

These types are also accessible directly as `typing.Pattern` and `typing.Match`.

class `typing.`**`NamedTuple`**

Typed version of namedtuple.

Usage:

```
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 two extra attributes: `_field_types`, giving a dict mapping field names to types, and `_field_defaults`, a dict mapping field names to default values. (The field names are in the `_fields` attribute, which is 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}>'
```

Backward-compatible usage:

```
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 doc-strings.

`typing.`**`NewType`**(*typ*)

A helper function to indicate a distinct types to a typechecker, see NewType. At runtime it returns a function that returns its argument. Usage:

```
UserId = NewType('UserId', int)
first_user = UserId(1)
```

New in version 3.5.2.

`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.`**`get_type_hints`**(*obj*[, *globals*[, *locals*]])

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

This is often the same as `obj.__annotations__`. In addition, forward references encoded as string literals are handled by evaluating them in `globals` and `locals` namespaces. If necessary, `Optional[t]` is added for function and method annotations if a default value equal to `None` is set. For a class `C`, return a dictionary constructed by merging all the `__annotations__` along `C.__mro__` in reverse order.

`@typing.`**`overload`**

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). The `@overload`-decorated definitions are for the benefit of the type checker only, since they will be overwritten by the non-`@overload`-decorated definition, while the latter is used at runtime but should be ignored by a type checker. At runtime, calling a `@overload`-decorated function directly will raise `NotImplementedError`. An example of overload that gives a more precise type than can be expressed using a union or a type variable:

```
@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>
```

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

@typing.**no_type_check**

Decorator to indicate that annotations are not type hints.

This works as class or function decorator. With a class, it applies recursively to all methods defined in that class (but not to methods defined in its superclasses or subclasses).

This mutates the function(s) 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.**Any**

Special type indicating an unconstrained type.

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

typing.**NoReturn**

Special type indicating that a function never returns. For example:

```
from typing import NoReturn

def stop() -> NoReturn:
    raise RuntimeError('no way')
```

New in version 3.6.5.

typing.**Union**

Union type; `Union[X, Y]` means either X or Y.

To define a union, use e.g. `Union[int, str]`. Details:

- The arguments must be types and there must be at least one.
- 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]
```

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

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

- When a class and its subclass are present, the latter is skipped, e.g.:

```
Union[int, object] == object
```

- You cannot subclass or instantiate a union.
- You cannot write `Union[X][Y]`.
- You can use `Optional[X]` as a shorthand for `Union[X, None]`.

typing.**Optional**

Optional type.

`Optional[X]` is equivalent to `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 needn't use the `Optional` qualifier on its type annotation (although it is inferred if the default is `None`). A mandatory argument may still have an `Optional` type if an explicit value of `None` is allowed.

typing.**Tuple**

Tuple type; `Tuple[X, Y]` is the type of a tuple of two items with the first item of type X and the second of type Y.

Example: `Tuple[T1, T2]` is a tuple of two elements corresponding to type variables T1 and T2. `Tuple[int, float, str]` is a tuple of an int, a float and a string.

To specify a variable-length tuple of homogeneous type, use literal ellipsis, e.g. `Tuple[int, ...]`. A plain `Tuple` is equivalent to `Tuple[Any, ...]`, and in turn to `tuple`.

typing.**Callable**

Callable type; `Callable[[int], str]` is a function of (int) -> str.

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 or an ellipsis; the return type must be a single type.

There is no syntax to indicate optional or keyword arguments; such function types are rarely used as callback types. `Callable[..., ReturnType]` (literal ellipsis) can be used to type hint a callable taking any number of arguments and returning `ReturnType`. A plain `Callable` is equivalent to `Callable[..., Any]`, and in turn to `collections.abc.Callable`.

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 instanc
Starship.stats = {}     # This is OK
```

New in version 3.5.3.

`typing.`**AnyStr**

`AnyStr` is a type variable defined as `AnyStr = TypeVar('AnyStr', str, bytes)`.

It is meant to be used for functions that may accept any kind of string without allowing different kinds of strings to mix. For example:

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

concat(u"foo", u"bar")  # Ok, output has type 'unicode'
concat(b"foo", b"bar")  # Ok, output has type 'bytes'
concat(u"foo", b"bar")  # Error, cannot mix unicode and bytes
```

`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()
```

Note that 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.

New in version 3.5.2.