26.1. `typing` — Support for type hints

New in version 3.5.

Source code: Lib/typing.py

This module supports type hints as specified by **PEP 484**. The most fundamental support consists of the type `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:

```
Vector = List[float]
```

26.1.2. 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]`. `None` as a type hint is a special case and is replaced by `type(None)`.

26.1.3. 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]
```

26.1.4. 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('{}: {}'.format(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`:

```
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:

```
from typing import TypeVar, Mapping

T = TypeVar('T')

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

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

Subclassing 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 typing import Iterable

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

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.

26.1.5. The `Any` type

A special kind of type is `Any`. Every type is a subtype of `Any`. This is also true for the builtin type object. However, to the static type checker these are completely different.

When the type of a value is `object`, the 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. On the other hand, when a value has type `Any`, the type checker will allow all operations on it, and a value of type `Any` can be assigned to a variable (or used as a return value) of a more constrained type.

26.1.6. Classes, functions, and decorators

The module defines the following classes, functions and decorators:

class `typing.`**Any**

Special type indicating an unconstrained type.

- Any object is an instance of `Any`.
- Any class is a subclass of `Any`.
- As a special case, `Any` and `object` are subclasses of each other.

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.`**`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]
  ```

- If `Any` is present it is the sole survivor, e.g.:

  ```
  Union[int, Any] == Any
  ```

- 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]`.

class `typing.`**`Optional`**

Optional type.

`Optional[X]` is equivalent to `Union[X, type(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.

class `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, ...]`.

class `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; 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]` could be used to type hint a callable taking any number of arguments and returning `ReturnType`. A plain `Callable` is equivalent to `Callable[..., Any]`.

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

A generic version of the collections.abc.Iterable.

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

A generic version of the collections.abc.Iterator.

class typing.**SupportsInt**

An ABC with one abstract method __int__.

class typing.**SupportsFloat**

An ABC with one abstract method __float__.

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.**Reversible**

An ABC with one abstract method __reversed__ returning an Iterator[T_co].

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

A generic version of collections.abc.Container.

class typing.**AbstractSet**(*Sized, Iterable[T_co], Container[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, Iterable[KT], Container[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`**(*Sized, Iterable[T_co], Container[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.`**`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 slice__to_4(vector: Sequence[T]) -> List[T]:
    return vector[0:4]
```

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

A generic version of `builtins.set`.

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.`**`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) -> int:
    return word_list[word]
```

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

class typing.**io**

Wrapper namespace for I/O stream types.

This defines the generic type `IO[AnyStr]` and aliases `TextIO` and `BinaryIO` for respectively `IO[str]` and `IO[bytes]`. These representing the types of I/O streams such as returned by `open()`.

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]`.

typing.**NamedTuple**(*typename*, *fields*)

Typed version of namedtuple.

Usage:

```
Employee = typing.NamedTuple('Employee', [('name', str), ('id', int)])
```

This is equivalent to:

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

The resulting class has one extra attribute: _field_types, giving a dict mapping field names to types. (The field names are in the _fields attribute, which is part of the namedtuple API.)

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*)

Return type hints for a function or method object.

This is often the same as `obj.__annotations__`, but it handles forward references encoded as string literals, and if necessary adds `Optional[t]` if a default value equal to None is set.

@typing.**no_type_check**(*arg*)

Decorator to indicate that annotations are not type hints.

The argument must be a class or function; if it is 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*)

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()`.