PEP 484 – Type Hints

Author: Guido van Rossum <guido at python.org>, Jukka Lehtosalo <jukka.lehtosalo at iki.fi>, Łukasz Langa <lukasz at python.org>

3DFL-Delegate: Mark Shannon

Discussions-To: Python-Dev list

Status: Final

Type: Standards Track

Topic: Typing

Created: 29-Sep-2014

ython-Version: 3.5

Post-History: 16-Jan-2015, 20-Mar-2015, 17-Apr-2015, 20-May-2015, 22-May-2015

Resolution: Python-Dev message

---

bstract

:P 3107 introduced syntax for function annotations, but the semantics were deliberately lef ıdefined. There has now been enough 3rd party usage for static type analysis that the ımmunity would benefit from a standard vocabulary and baseline tools within the standarc ›rary.

ıis PEP introduces a provisional module to provide these standard definitions and tools, ɔng with some conventions for situations where annotations are not available.

ɔte that this PEP still explicitly does NOT prevent other uses of annotations, nor does it quire (or forbid) any particular processing of annotations, even when they conform to this ›ecification. It simply enables better coordination, as PEP 333 did for web frameworks.

›r example, here is a simple function whose argument and return type are declared in the ınotations:

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

'hile these annotations are available at runtime through the usual `__annotations__` attribute › *type checking happens at runtime*. Instead, the proposal assumes the existence of a ›parate off-line type checker which users can run over their source code voluntarily.

sentially, such a type checker acts as a very powerful linter. (While it would of course be ossible for individual users to employ a similar checker at run time for Design By Contract nforcement or JIT optimization, those tools are not yet as mature.)

e proposal is strongly inspired by mypy. For example, the type "sequence of integers" can written as `Sequence[int]`. The square brackets mean that no new syntax needs to be adde the language. The example here uses a custom type `Sequence`, imported from a pure-thon module `typing`. The `Sequence[int]` notation works at runtime by implementing `getitem__()` in the metaclass (but its significance is primarily to an offline type checker).

e type system supports unions, generic types, and a special type named `Any` which is nsistent with (i.e. assignable to and from) all types. This latter feature is taken from the ide gradual typing. Gradual typing and the full type system are explained in PEP 483.

ther approaches from which we have borrowed or to which ours can be compared and ntrasted are described in PEP 482.

ationale and Goals

P 3107 added support for arbitrary annotations on parts of a function definition. Althoug meaning was assigned to annotations then, there has always been an implicit goal to use em for type hinting, which is listed as the first possible use case in said PEP.

is PEP aims to provide a standard syntax for type annotations, opening up Python code to sier static analysis and refactoring, potential runtime type checking, and (perhaps, in some ntexts) code generation utilizing type information.

f these goals, static analysis is the most important. This includes support for off-line type eckers such as mypy, as well as providing a standard notation that can be used by IDEs fo de completion and refactoring.

on-goals

hile the proposed typing module will contain some building blocks for runtime type ecking – in particular the `get_type_hints()` function – third party packages would have to veloped to implement specific runtime type checking functionality, for example using corators or metaclasses. Using type hints for performance optimizations is left as an ercise for the reader.

should also be emphasized that **Python will remain a dynamically typed language, and e authors have no desire to ever make type hints mandatory, even by convention.**

ıe meaning of annotations

ıy function without annotations should be treated as having the most general type possibl ignored, by any type checker. Functions with the `@no_type_check` decorator should be eated as having no annotations.

is recommended but not required that checked functions have annotations for all guments and the return type. For a checked function, the default annotation for argument ıd for the return type is `Any`. An exception is the first argument of instance and class ethods. If it is not annotated, then it is assumed to have the type of the containing class fo stance methods, and a type object type corresponding to the containing class object for ass methods. For example, in class `A` the first argument of an instance method has the ıplicit type `A`. In a class method, the precise type of the first argument cannot be presented using the available type notation.

lote that the return type of `__init__` ought to be annotated with `-> None`. The reason for is is subtle. If `__init__` assumed a return annotation of `-> None`, would that mean that an gument-less, un-annotated `__init__` method should still be type-checked? Rather than aving this ambiguous or introducing an exception to the exception, we simply say that `_init__` ought to have a return annotation; the default behavior is thus the same as for othe ethods.)

type checker is expected to check the body of a checked function for consistency with the ven annotations. The annotations may also be used to check correctness of calls appearinc other checked functions.

pe checkers are expected to attempt to infer as much information as necessary. The inimum requirement is to handle the builtin decorators `@property`, `@staticmethod` and `:lassmethod`.

ype Definition Syntax

ıe syntax leverages PEP 3107-style annotations with a number of extensions described in ctions below. In its basic form, type hinting is used by filling function annotation slots with asses:

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

ıis states that the expected type of the `name` argument is `str`. Analogically, the expected turn type is `str`.

:pressions whose type is a subtype of a specific argument type are also accepted for that gument.

:ceptable type hints

'pe hints may be built-in classes (including those defined in standard library or third-party :tension modules), abstract base classes, types available in the `types` module, and user-:fined classes (including those defined in the standard library or third-party modules).

'hile annotations are normally the best format for type hints, there are times when it is mor ›propriate to represent them by a special comment, or in a separately distributed stub file. ee below for examples.)

ınotations must be valid expressions that evaluate without raising exceptions at the time e function is defined (but see below for forward references).

ınotations should be kept simple or static analysis tools may not be able to interpret the ılues. For example, dynamically computed types are unlikely to be understood. (This is an tentionally somewhat vague requirement, specific inclusions and exclusions may be added future versions of this PEP as warranted by the discussion.)

addition to the above, the following special constructs defined below may be used: `None`, `ıy`, `Union`, `Tuple`, `Callable`, all ABCs and stand-ins for concrete classes exported from `typing` .g. `Sequence` and `Dict`), type variables, and type aliases.

l newly introduced names used to support features described in following sections (such a `ıy` and `Union`) are available in the `typing` module.

sing None

'hen used in a type hint, the expression `None` is considered equivalent to `type(None)`.

'pe aliases

'pe aliases are defined by simple variable assignments:

```
Jrl = str

def retry(url: Url, retry_count: int) -> None: ...
```

ote that we recommend capitalizing alias names, since they represent user-defined types, ıich (like user-defined classes) are typically spelled that way.

pe aliases may be as complex as type hints in annotations – anything that is acceptable as pe hint is acceptable in a type alias:

```
from typing import TypeVar, Iterable, Tuple

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

def inproduct(v: Vector[T]) -> T:
    return sum(x*y for x, y in v)
def dilate(v: Vector[T], scale: T) -> Vector[T]:
    return ((x * scale, y * scale) for x, y in v)
vec = []  # type: Vector[float]
```

is is equivalent to:

```
from typing import TypeVar, Iterable, Tuple

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

def inproduct(v: Iterable[Tuple[T, T]]) -> T:
    return sum(x*y for x, y in v)
def dilate(v: Iterable[Tuple[T, T]], scale: T) -> Iterable[Tuple[T, T]]:
    return ((x * scale, y * scale) for x, y in v)
vec = []  # type: Iterable[Tuple[float, float]]
```

allable

ameworks expecting callback functions of specific signatures might be type hinted using `llable[[Arg1Type, Arg2Type], ReturnType]`. Examples:

```
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
```

is possible to declare the return type of a callable without specifying the call signature by bstituting a literal ellipsis (three dots) for the list of arguments:

```
def partial(func: Callable[..., str], *args) -> Callable[..., str]:
    # Body
```

ote that there are no square brackets around the ellipsis. The arguments of the callback are mpletely unconstrained in this case (and keyword arguments are acceptable).

nce using callbacks with keyword arguments is not perceived as a common use case, there currently no support for specifying keyword arguments with `Callable`. Similarly, there is nc ipport for specifying callback signatures with a variable number of arguments of a specific pe.

ecause `typing.Callable` does double-duty as a replacement for `collections.abc.Callable`, `instance(x, typing.Callable)` is implemented by deferring to `isinstance(x, llections.abc.Callable)`. However, `isinstance(x, typing.Callable[...])` is not supported.

enerics

nce type information about objects kept in containers cannot be statically inferred in a eneric way, abstract base classes have been extended to support subscription to denote pected types for container elements. Example:

```
from typing import Mapping, Set

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

enerics can be parameterized by using a new factory available in `typing` called `TypeVar`. ample:

```
from typing import Sequence, TypeVar

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

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

this case the contract is that the returned value is consistent with the elements held by the llection.

`TypeVar()` expression must always directly be assigned to a variable (it should not be used part of a larger expression). The argument to `TypeVar()` must be a string equal to the riable name to which it is assigned. Type variables must not be redefined.

`peVar` supports constraining parametric types to a fixed set of possible types (note: those pes cannot be parameterized by type variables). For example, we can define a type variable at ranges over just `str` and `bytes`. By default, a type variable ranges over all possible types ample of constraining a type variable:

```
from typing import TypeVar, Text

AnyStr = TypeVar('AnyStr', Text, bytes)

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

e function `concat` can be called with either two `str` arguments or two `bytes` arguments, b
t with a mix of `str` and `bytes` arguments.

ere should be at least two constraints, if any; specifying a single constraint is disallowed.

btypes of types constrained by a type variable should be treated as their respective
plicitly listed base types in the context of the type variable. Consider this example:

```
class MyStr(str): ...

x = concat(MyStr('apple'), MyStr('pie'))
```

e call is valid but the type variable `AnyStr` will be set to `str` and not `MyStr`. In effect, the
ferred type of the return value assigned to `x` will also be `str`.

dditionally, `Any` is a valid value for every type variable. Consider the following:

```
def count_truthy(elements: List[Any]) -> int:
    return sum(1 for elem in elements if elem)
```

is is equivalent to omitting the generic notation and just saying `elements: List`.

ser-defined generic types

u can include a `Generic` base class to define a user-defined class as generic. Example:

```
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. This is valid:

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

T = TypeVar('T')
S = TypeVar('S')

class Pair(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
    ...
```

he `Generic[T]` base class is redundant in simple cases where you subclass some other eneric class and specify type variables for its parameters:

```
from typing import TypeVar, Iterator

T = TypeVar('T')

class MyIter(Iterator[T]):
    ...
```

hat class definition is equivalent to:

```
class MyIter(Iterator[T], Generic[T]):
    ...
```

ou can use multiple inheritance with `Generic`:

```
from typing import TypeVar, Generic, Sized, Iterable, Container, Tuple

T = TypeVar('T')

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

K = TypeVar('K')
V = TypeVar('V')

class MyMapping(Iterable[Tuple[K, V]],
                Container[Tuple[K, V]],
                Generic[K, V]):
    ...
```

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

```
from typing import Iterable

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

eneric metaclasses are not supported.

:oping rules for type variables

'pe variables follow normal name resolution rules. However, there are some special cases ir
e static typechecking context:

- A type variable used in a generic function could be inferred to represent different type in the same code block. Example:

```
from typing import TypeVar, Generic

T = TypeVar('T')

def fun_1(x: T) -> T: ...  # T here
def fun_2(x: T) -> T: ...  # and here could be different

fun_1(1)                   # This is OK, T is inferred to be int
fun_2('a')                 # This is also OK, now T is str
```

- A type variable used in a method of a generic class that coincides with one of the variables that parameterize this class is always bound to that variable. Example:

```
from typing import TypeVar, Generic

T = TypeVar('T')

class MyClass(Generic[T]):
    def meth_1(self, x: T) -> T: ...  # T here
    def meth_2(self, x: T) -> T: ...  # and here are always the same

a = MyClass()  # type: MyClass[int]
a.meth_1(1)    # OK
a.meth_2('a')  # This is an error!
```

- A type variable used in a method that does not match any of the variables that parameterize the class makes this method a generic function in that variable:

```
T = TypeVar('T')
S = TypeVar('S')
class Foo(Generic[T]):
    def method(self, x: T, y: S) -> S:
        ...

x = Foo()               # type: Foo[int]
y = x.method(0, "abc")  # inferred type of y is str
```

- Unbound type variables should not appear in the bodies of generic functions, or in the class bodies apart from method definitions:

```
T = TypeVar('T')
S = TypeVar('S')

def a_fun(x: T) -> None:
    # this is OK
    y = []  # type: List[T]
    # but below is an error!
    y = []  # type: List[S]

class Bar(Generic[T]):
    # this is also an error
    an_attr = []  # type: List[S]

    def do_something(x: S) -> S:  # this is OK though
        ...
```

- A generic class definition that appears inside a generic function should not use type variables that parameterize the generic function:

```
from typing import List

def a_fun(x: T) -> None:

    # This is OK
    a_list = []  # type: List[T]
    ...

    # This is however illegal
    class MyGeneric(Generic[T]):
        ...
```

- A generic class nested in another generic class cannot use same type variables. The scope of the type variables of the outer class doesn't cover the inner one:

```
T = TypeVar('T')
S = TypeVar('S')

class Outer(Generic[T]):
    class Bad(Iterable[T]):       # Error
        ...
    class AlsoBad:
        x = None  # type: List[T] # Also an error

    class Inner(Iterable[S]):     # OK
        ...
    attr = None  # type: Inner[T] # Also OK
```

stantiating generic classes and type erasure

ser-defined generic classes can be instantiated. Suppose we write a `Node` class inheriting om `Generic[T]`:

```
from typing import TypeVar, Generic

T = TypeVar('T')

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

create `Node` instances you call `Node()` just as for a regular class. At runtime the type (class) the instance will be `Node`. But what type does it have to the type checker? The answer epends on how much information is available in the call. If the constructor (`__init__` or `new__`) uses `T` in its signature, and a corresponding argument value is passed, the type of e corresponding argument(s) is substituted. Otherwise, `Any` is assumed. Example:

```
from typing import TypeVar, Generic

T = TypeVar('T')

class Node(Generic[T]):
    x = None  # type: T # Instance attribute (see below)
    def __init__(self, label: T = None) -> None:
        ...

x = Node('')  # Inferred type is Node[str]
y = Node(0)   # Inferred type is Node[int]
z = Node()    # Inferred type is Node[Any]
```

case the inferred type uses `[Any]` but the intended type is more specific, you can use a typ mment (see below) to force the type of the variable, e.g.:

```
# (continued from previous example)
a = Node()  # type: Node[int]
b = Node()  # type: Node[str]
```

ternatively, you can instantiate a specific concrete type, e.g.:

```
# (continued from previous example)
p = Node[int]()
q = Node[str]()
r = Node[int]('')  # Error
s = Node[str](0)   # Error
```

ote that the runtime type (class) of `p` and `q` is still just `Node` – `Node[int]` and `Node[str]` are stinguishable class objects, but the runtime class of the objects created by instantiating em doesn't record the distinction. This behavior is called "type erasure"; it is common actice in languages with generics (e.g. Java, TypeScript).

sing generic classes (parameterized or not) to access attributes will result in type check ilure. Outside the class definition body, a class attribute cannot be assigned, and can only e looked up by accessing it through a class instance that does not have an instance attribut th the same name:

```
# (continued from previous example)
Node[int].x = 1  # Error
Node[int].x      # Error
Node.x = 1       # Error
Node.x           # Error
type(p).x        # Error
p.x              # Ok (evaluates to None)
Node[int]().x    # Ok (evaluates to None)
p.x = 1          # Ok, but assigning to instance attribute
```

eneric versions of abstract collections like `Mapping` or `Sequence` and generic versions of built classes – `List`, `Dict`, `Set`, and `FrozenSet` – cannot be instantiated. However, concrete user-efined subclasses thereof and generic versions of concrete collections can be instantiated:

```
data = DefaultDict[int, bytes]()
```

ote that one should not confuse static types and runtime classes. The type is still erased in is case and the above expression is just a shorthand for:

```
data = collections.defaultdict()  # type: DefaultDict[int, bytes]
```

is not recommended to use the subscripted class (e.g. `Node[int]`) directly in an expression sing a type alias (e.g. `IntNode = Node[int]`) instead is preferred. (First, creating the bscripted class, e.g. `Node[int]`, has a runtime cost. Second, using a type alias is more adable.)

rbitrary generic types as base classes

`eneric[T]` is only valid as a base class – it's not a proper type. However, user-defined generi pes such as `LinkedList[T]` from the above example and built-in generic types and ABCs ich as `List[T]` and `Iterable[T]` are valid both as types and as base classes. For example, we n define a subclass of `Dict` that specializes type arguments:

```
from typing import Dict, List, Optional

class Node:
    ...

class SymbolTable(Dict[str, List[Node]]):
    def push(self, name: str, node: Node) -> None:
        self.setdefault(name, []).append(node)

    def pop(self, name: str) -> Node:
        return self[name].pop()

    def lookup(self, name: str) -> Optional[Node]:
        nodes = self.get(name)
        if nodes:
            return nodes[-1]
        return None
```

`mbolTable` is a subclass of `dict` and a subtype of `Dict[str, List[Node]]`.

a generic base class has a type variable as a type argument, this makes the defined class eneric. For example, we can define a generic `LinkedList` class that is iterable and a containe

```
from typing import TypeVar, Iterable, Container

T = TypeVar('T')

class LinkedList(Iterable[T], Container[T]):
    ...
```

ow `LinkedList[int]` is a valid type. Note that we can use `T` multiple times in the base class t, as long as we don't use the same type variable `T` multiple times within `Generic[...]`.

so consider the following example:

```
from typing import TypeVar, Mapping

T = TypeVar('T')

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

this case MyDict has a single parameter, T.

ostract generic types

ne metaclass used by `Generic` is a subclass of `abc.ABCMeta`. A generic class can be an ABC by cluding abstract methods or properties, and generic classes can also have ABCs as base asses without a metaclass conflict.

ype variables with an upper bound

type variable may specify an upper bound using `bound=<type>` (note: <type> itself cannot b
arameterized by type variables). This means that an actual type substituted (explicitly or
nplicitly) for the type variable must be a subtype of the boundary type. Example:

```
from typing import TypeVar, Sized

ST = TypeVar('ST', bound=Sized)

def longer(x: ST, y: ST) -> ST:
    if len(x) > len(y):
        return x
    else:
        return y

longer([1], [1, 2])  # ok, return type List[int]
longer({1}, {1, 2})  # ok, return type Set[int]
longer([1], {1, 2})  # ok, return type Collection[int]
```

n upper bound cannot be combined with type constraints (as in used `AnyStr`, see the
ample earlier); type constraints cause the inferred type to be _exactly_ one of the constrain
pes, while an upper bound just requires that the actual type is a subtype of the boundary
pe.

ovariance and contravariance

onsider a class `Employee` with a subclass `Manager`. Now suppose we have a function with an
gument annotated with `List[Employee]`. Should we be allowed to call this function with a
riable of type `List[Manager]` as its argument? Many people would answer "yes, of course"
thout even considering the consequences. But unless we know more about the function, a
pe checker should reject such a call: the function might append an `Employee` instance to th
t, which would violate the variable's type in the caller.

turns out such an argument acts *contravariantly*, whereas the intuitive answer (which is
orrect in case the function doesn't mutate its argument!) requires the argument to act
variantly. A longer introduction to these concepts can be found on Wikipedia and in PEP
33; here we just show how to control a type checker's behavior.

 default generic types are considered *invariant* in all type variables, which means that
lues for variables annotated with types like `List[Employee]` must exactly match the type
notation – no subclasses or superclasses of the type parameter (in this example `Employee`)
e allowed.

ı facilitate the declaration of container types where covariant or contravariant type checkin acceptable, type variables accept keyword arguments `covariant=True` or `contravariant=True` : most one of these may be passed. Generic types defined with such variables are ɔnsidered covariant or contravariant in the corresponding variable. By convention, it is commended to use names ending in `_co` for type variables defined with `covariant=True` an ımes ending in `_contra` for that defined with `contravariant=True`.

typical example involves defining an immutable (or read-only) container class:

```
from typing import TypeVar, Generic, Iterable, Iterator

T_co = TypeVar('T_co', covariant=True)

class ImmutableList(Generic[T_co]):
    def __init__(self, items: Iterable[T_co]) -> None: ...
    def __iter__(self) -> Iterator[T_co]: ...
    ...

class Employee: ...

class Manager(Employee): ...

def dump_employees(emps: ImmutableList[Employee]) -> None:
    for emp in emps:
        ...

mgrs = ImmutableList([Manager()])  # type: ImmutableList[Manager]
dump_employees(mgrs)  # OK
```

ıe read-only collection classes in `typing` are all declared covariant in their type variable (e.g ıpping and `Sequence`). The mutable collection classes (e.g. `MutableMapping` and ıtableSequence) are declared invariant. The one example of a contravariant type is the ːnerator type, which is contravariant in the `send()` argument type (see below).

ɔte: Covariance or contravariance is *not* a property of a type variable, but a property of a ːneric class defined using this variable. Variance is only applicable to generic types; generic nctions do not have this property. The latter should be defined using only type variables thout `covariant` or `contravariant` keyword arguments. For example, the following example fine:

```
from typing import TypeVar

class Employee: ...

class Manager(Employee): ...

E = TypeVar('E', bound=Employee)

def dump_employee(e: E) -> None: ...

dump_employee(Manager())  # OK
```

hile the following is prohibited:

```
B_co = TypeVar('B_co', covariant=True)

def bad_func(x: B_co) -> B_co:  # Flagged as error by a type checker
    ...
```

ne numeric tower

:P 3141 defines Python's numeric tower, and the stdlib module `numbers` implements the orresponding ABCs (`Number`, `Complex`, `Real`, `Rational` and `Integral`). There are some issues th these ABCs, but the built-in concrete numeric classes `complex`, `float` and `int` are oiquitous (especially the latter two :-).

ather than requiring that users write `import numbers` and then use `numbers.Float` etc., this PE oposes a straightforward shortcut that is almost as effective: when an argument is nnotated as having type `float`, an argument of type `int` is acceptable; similar, for an gument annotated as having type `complex`, arguments of type `float` or `int` are acceptable nis does not handle classes implementing the corresponding ABCs or the `fractions.Fractic` ass, but we believe those use cases are exceedingly rare.

orward references

hen a type hint contains names that have not been defined yet, that definition may be :pressed as a string literal, to be resolved later.

situation where this occurs commonly is the definition of a container class, where the class eing defined occurs in the signature of some of the methods. For example, the following ode (the start of a simple binary tree implementation) does not work:

```
class Tree:
    def __init__(self, left: Tree, right: Tree):
        self.left = left
        self.right = right
```

› address this, we write:

```
class Tree:
    def __init__(self, left: 'Tree', right: 'Tree'):
        self.left = left
        self.right = right
```

ıe string literal should contain a valid Python expression (i.e., `compile(lit, '', 'eval')` ıould be a valid code object) and it should evaluate without errors once the module has ꜫen fully loaded. The local and global namespace in which it is evaluated should be the me namespaces in which default arguments to the same function would be evaluated.

oreover, the expression should be parseable as a valid type hint, i.e., it is constrained by th les from the section Acceptable type hints above.

is allowable to use string literals as *part* of a type hint, for example:

```
class Tree:
    ...
    def leaves(self) -> List['Tree']:
        ...
```

common use for forward references is when e.g. Django models are needed in the gnatures. Typically, each model is in a separate file, and has methods taking arguments ıose type involves other models. Because of the way circular imports work in Python, it is ten not possible to import all the needed models directly:

```
# File models/a.py
from models.b import B
class A(Model):
    def foo(self, b: B): ...

# File models/b.py
from models.a import A
class B(Model):
    def bar(self, a: A): ...

# File main.py
from models.a import A
from models.b import B
```

ssuming main is imported first, this will fail with an ImportError at the line `from models.a port A` in models/b.py, which is being imported from models/a.py before a has defined ass A. The solution is to switch to module-only imports and reference the models by their ıodule_._class_ name:

```
# File models/a.py
from models import b
class A(Model):
    def foo(self, b: 'b.B'): ...

# File models/b.py
from models import a
class B(Model):
    def bar(self, a: 'a.A'): ...

# File main.py
from models.a import A
from models.b import B
```

nion types

nce accepting a small, limited set of expected types for a single argument is common, ther a new special factory called `Union`. Example:

```
from typing import Union

def handle_employees(e: Union[Employee, Sequence[Employee]]) -> None:
    if isinstance(e, Employee):
        e = [e]
    ...
```

type factored by `Union[T1, T2, ...]` is a supertype of all types `T1`, `T2`, etc., so that a value at is a member of one of these types is acceptable for an argument annotated by `Union[T1 , ...]`.

ne common case of union types are *optional* types. By default, `None` is an invalid value for ıy type, unless a default value of `None` has been provided in the function definition. :amples:

```
def handle_employee(e: Union[Employee, None]) -> None: ...
```

; a shorthand for `Union[T1, None]` you can write `Optional[T1]`; for example, the above is ʲuivalent to:

```
from typing import Optional

def handle_employee(e: Optional[Employee]) -> None: ...
```

past version of this PEP allowed type checkers to assume an optional type when the defau ılue is `None`, as in this code:

```
def handle_employee(e: Employee = None): ...
```

ıis would have been treated as equivalent to:

```
def handle_employee(e: Optional[Employee] = None) -> None: ...
```

ıis is no longer the recommended behavior. Type checkers should move towards requiring e optional type to be made explicit.

ıpport for singleton types in unions

singleton instance is frequently used to mark some special condition, in particular in :uations where `None` is also a valid value for a variable. Example:

```
_empty = object()

def func(x=_empty):
    if x is _empty:  # default argument value
        return 0
    elif x is None:  # argument was provided and it's None
        return 1
    else:
        return x * 2
```

allow precise typing in such situations, the user should use the `Union` type in conjunction th the `enum.Enum` class provided by the standard library, so that type errors can be caught atically:

```
from typing import Union
from enum import Enum

class Empty(Enum):
    token = 0
_empty = Empty.token

def func(x: Union[int, None, Empty] = _empty) -> int:

    boom = x * 42  # This fails type check

    if x is _empty:
        return 0
    elif x is None:
        return 1
    else:  # At this point typechecker knows that x can only have type int
        return x * 2
```

nce the subclasses of `Enum` cannot be further subclassed, the type of variable `x` can be atically inferred in all branches of the above example. The same approach is applicable if ore than one singleton object is needed: one can use an enumeration that has more than ıe value:

```
class Reason(Enum):
    timeout = 1
    error = 2

def process(response: Union[str, Reason] = '') -> str:
    if response is Reason.timeout:
        return 'TIMEOUT'
    elif response is Reason.error:
        return 'ERROR'
    else:
        # response can be only str, all other possible values exhausted
        return 'PROCESSED: ' + response
```

ıe `Any` type

special kind of type is `Any`. Every type is consistent with `Any`. It can be considered a type at has all values and all methods. Note that `Any` and builtin type `object` are completely fferent.

'hen the type of a value is `object`, the type checker will reject almost all operations on it, ar signing it to a variable (or using it as a return value) of a more specialized type is a type ror. On the other hand, when a value has type `Any`, the type checker will allow all operatior ı it, and a value of type `Any` can be assigned to a variable (or used as a return value) of a ore constrained type.

function parameter without an annotation is assumed to be annotated with `Any`. If a gener pe is used without specifying type parameters, they are assumed to be `Any`:

```
from typing import Mapping

def use_map(m: Mapping) -> None:  # Same as Mapping[Any, Any]
    ...
```

ıis rule also applies to `Tuple`, in annotation context it is equivalent to `Tuple[Any, ...]` and, rn, to `tuple`. As well, a bare `Callable` in an annotation is equivalent to `Callable[..., Any]` ıd, in turn, to `collections.abc.Callable`:

```
from typing import Tuple, List, Callable

def check_args(args: Tuple) -> bool:
    ...

check_args(())           # OK
check_args((42, 'abc'))  # Also OK
check_args(3.14)         # Flagged as error by a type checker

# A list of arbitrary callables is accepted by this function
def apply_callbacks(cbs: List[Callable]) -> None:
    ...
```

he `NoReturn` type

he `typing` module provides a special type `NoReturn` to annotate functions that never return ormally. For example, a function that unconditionally raises an exception:

```
from typing import NoReturn

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

he `NoReturn` annotation is used for functions such as `sys.exit`. Static type checkers will nsure that functions annotated as returning `NoReturn` truly never return, either implicitly or xplicitly:

```
import sys
from typing import NoReturn

  def f(x: int) -> NoReturn:  # Error, f(0) implicitly returns None
      if x != 0:
          sys.exit(1)
```

he checkers will also recognize that the code after calls to such functions is unreachable an ll behave accordingly:

```
# continue from first example
def g(x: int) -> int:
    if x > 0:
        return x
    stop()
    return 'whatever works'  # Error might be not reported by some checkers
                             # that ignore errors in unreachable blocks
```

he `NoReturn` type is only valid as a return annotation of functions, and considered an error i appears in other positions:

```
from typing import List, NoReturn

# All of the following are errors
def bad1(x: NoReturn) -> int:
    ...
bad2 = None  # type: NoReturn
def bad3() -> List[NoReturn]:
    ...
```

e type of class objects

ometimes you want to talk about class objects, in particular class objects that inherit from a ven class. This can be spelled as `Type[C]` where `C` is a class. To clarify: while `C` (when used a annotation) refers to instances of class `C`, `Type[C]` refers to *subclasses* of `C`. (This is a simil stinction as between `object` and `type`.)

r example, suppose we have the following classes:

```
class User: ...  # Abstract base for User classes
class BasicUser(User): ...
class ProUser(User): ...
class TeamUser(User): ...
```

d suppose we have a function that creates an instance of one of these classes if you pass class object:

```
def new_user(user_class):
    user = user_class()
    # (Here we could write the user object to a database)
    return user
```

ithout `Type[]` the best we could do to annotate `new_user()` would be:

```
def new_user(user_class: type) -> User:
    ...
```

owever using `Type[]` and a type variable with an upper bound we can do much better:

```
U = TypeVar('U', bound=User)
def new_user(user_class: Type[U]) -> U:
    ...
```

ow when we call `new_user()` with a specific subclass of `User` a type checker will infer the rrect type of the result:

```
joe = new_user(BasicUser)  # Inferred type is BasicUser
```

ıe value corresponding to `Type[C]` must be an actual class object that's a subtype of `C`, not ›ecial form. In other words, in the above example calling e.g. `new_user(Union[BasicUser, oUser])` is rejected by the type checker (in addition to failing at runtime because you can't stantiate a union).

ɔte that it is legal to use a union of classes as the parameter for `Type[]`, as in:

```
def new_non_team_user(user_class: Type[Union[BasicUser, ProUser]]):
    user = new_user(user_class)
    ...
```

ɔwever the actual argument passed in at runtime must still be a concrete class object, e.g. e above example:

```
new_non_team_user(ProUser)  # OK
new_non_team_user(TeamUser)  # Disallowed by type checker
```

`pe[Any]` is also supported (see below for its meaning).

`pe[T]` where `T` is a type variable is allowed when annotating the first argument of a class ethod (see the relevant section).

ıy other special constructs like `Tuple` or `Callable` are not allowed as an argument to `Type`.

ıere are some concerns with this feature: for example when `new_user()` calls `user_class()` is implies that all subclasses of `User` must support this in their constructor signature. ɔwever this is not unique to `Type[]`: class methods have similar concerns. A type checker ıght to flag violations of such assumptions, but by default constructor calls that match the ›nstructor signature in the indicated base class (`User` in the example above) should be lowed. A program containing a complex or extensible class hierarchy might also handle thi ' using a factory class method. A future revision of this PEP may introduce better ways of ealing with these concerns.

'hen `Type` is parameterized it requires exactly one parameter. Plain `Type` without brackets is ıuivalent to `Type[Any]` and this in turn is equivalent to `type` (the root of Python's metaclass erarchy). This equivalence also motivates the name, `Type`, as opposed to alternatives like `ass` or `SubType`, which were proposed while this feature was under discussion; this is simila the relationship between e.g. `List` and `list`.

egarding the behavior of `Type[Any]` (or `Type` or `type`), accessing attributes of a variable with is type only provides attributes and methods defined by `type` (for example, `__repr__()` and `mro__`). Such a variable can be called with arbitrary arguments, and the return type is `Any`.

'pe is covariant in its parameter, because `Type[Derived]` is a subtype of `Type[Base]`:

```
def new_pro_user(pro_user_class: Type[ProUser]):
    user = new_user(pro_user_class)  # OK
    ...
```

ınotating instance and class methods

most cases the first argument of class and instance methods does not need to be ınotated, and it is assumed to have the type of the containing class for instance methods, ıd a type object type corresponding to the containing class object for class methods. In ldition, the first argument in an instance method can be annotated with a type variable. In is case the return type may use the same type variable, thus making that method a generi nction. For example:

```
T = TypeVar('T', bound='Copyable')
class Copyable:
    def copy(self: T) -> T:
        # return a copy of self

class C(Copyable): ...
c = C()
c2 = c.copy()  # type here should be C
```

ıe same applies to class methods using `Type[]` in an annotation of the first argument:

```
T = TypeVar('T', bound='C')
class C:
    @classmethod
    def factory(cls: Type[T]) -> T:
        # make a new instance of cls

class D(C): ...
d = D.factory()  # type here should be D
```

ote that some type checkers may apply restrictions on this use, such as requiring an opropriate upper bound for the type variable used (see examples).

ersion and platform checking

'pe checkers are expected to understand simple version and platform checks, e.g.:

```
import sys

if sys.version_info[0] >= 3:
    # Python 3 specific definitions
else:
    # Python 2 specific definitions

if sys.platform == 'win32':
    # Windows specific definitions
else:
    # Posix specific definitions
```

on't expect a checker to understand obfuscations like `"".join(reversed(sys.platform)) == unil"`.

untime or type checking?

ometimes there's code that must be seen by a type checker (or other static analysis tools) ut should not be executed. For such situations the `typing` module defines a constant, `PE_CHECKING`, that is considered `True` during type checking (or other static analysis) but `Fal` runtime. Example:

```
import typing

if typing.TYPE_CHECKING:
    import expensive_mod

def a_func(arg: 'expensive_mod.SomeClass') -> None:
    a_var = arg  # type: expensive_mod.SomeClass
    ...
```

lote that the type annotation must be enclosed in quotes, making it a "forward reference", hide the `expensive_mod` reference from the interpreter runtime. In the `# type` comment no uotes are needed.)

is approach may also be useful to handle import cycles.

bitrary argument lists and default argument values

bitrary argument lists can as well be type annotated, so that the definition:

```
def foo(*args: str, **kwds: int): ...
```

acceptable and it means that, e.g., all of the following represent function calls with valid pes of arguments:

```
foo('a', 'b', 'c')
foo(x=1, y=2)
foo('', z=0)
```

the body of function `foo`, the type of variable `args` is deduced as `Tuple[str, ...]` and the pe of variable `kwds` is `Dict[str, int]`.

stubs it may be useful to declare an argument as having a default without specifying the :tual default value. For example:

```
def foo(x: AnyStr, y: AnyStr = ...) -> AnyStr: ...
```

'hat should the default value look like? Any of the options `""`, `b""` or `None` fails to satisfy th pe constraint.

such cases the default value may be specified as a literal ellipsis, i.e. the above example is erally what you would write.

›sitional-only arguments

›me functions are designed to take their arguments only positionally, and expect their ıllers never to use the argument's name to provide that argument by keyword. All guments with names beginning with __ are assumed to be positional-only, except if their ımes also end with __:

```
def quux(__x: int, __y__: int = 0) -> None: ...

quux(3, __y__=1)  # This call is fine.

quux(__x=3)  # This call is an error.
```

ınotating generator functions and coroutines

ıe return type of generator functions can be annotated by the generic type `:nerator[yield_type, send_type, return_type]` provided by `typing.py` module:

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

oroutines introduced in PEP 492 are annotated with the same syntax as ordinary functions. owever, the return type annotation corresponds to the type of `await` expression, not to the oroutine type:

```
async def spam(ignored: int) -> str:
    return 'spam'

async def foo() -> None:
    bar = await spam(42)  # type: str
```

ne `typing.py` module provides a generic version of ABC `collections.abc.Coroutine` to specif vaitables that also support `send()` and `throw()` methods. The variance and order of type iriables correspond to those of `Generator`, namely `Coroutine[T_co, T_contra, V_co]`, for ample:

```
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
```

ne module also provides generic ABCs `Awaitable`, `AsyncIterable`, and `AsyncIterator` for tuations where more precise types cannot be specified:

```
def op() -> typing.Awaitable[str]:
    if cond:
        return spam(42)
    else:
        return asyncio.Future(...)
```

ompatibility with other uses of function annotations

number of existing or potential use cases for function annotations exist, which are compatible with type hinting. These may confuse a static type checker. However, since typ nting annotations have no runtime behavior (other than evaluation of the annotation pression and storing annotations in the `__annotations__` attribute of the function object), is does not make the program incorrect – it just may cause a type checker to emit spuriou arnings or errors.

mark portions of the program that should not be covered by type hinting, you can use or more of the following:

- a `# type: ignore` comment;
- a `@no_type_check` decorator on a class or function;

- a custom class or function decorator marked with `@no_type_check_decorator`.

›r more details see later sections.

order for maximal compatibility with offline type checking it may eventually be a good ide change interfaces that rely on annotations to switch to a different mechanism, for exampl decorator. In Python 3.5 there is no pressure to do this, however. See also the longer scussion under Rejected alternatives below.

ype comments

o first-class syntax support for explicitly marking variables as being of a specific type is lded by this PEP. To help with type inference in complex cases, a comment of the followinc rmat may be used:

```
x = []                # type: List[Employee]
x, y, z = [], [], []  # type: List[int], List[int], List[str]
x, y, z = [], [], []  # type: (List[int], List[int], List[str])
a, b, *c = range(5)   # type: float, float, List[float]
x = [1, 2]            # type: List[int]
```

pe comments should be put on the last line of the statement that contains the variable efinition. They can also be placed on `with` statements and `for` statements, right after the olon.

amples of type comments on `with` and `for` statements:

```
with frobnicate() as foo:  # type: int
    # Here foo is an int
    ...

for x, y in points:  # type: float, float
    # Here x and y are floats
    ...
```

stubs it may be useful to declare the existence of a variable without giving it an initial ılue. This can be done using PEP 526 variable annotation syntax:

```
from typing import IO

stream: IO[str]
```

ıe above syntax is acceptable in stubs for all versions of Python. However, in non-stub cod r versions of Python 3.5 and earlier there is a special case:

```
from typing import IO

stream = None  # type: IO[str]
```

ype checkers should not complain about this (despite the value `None` not matching the give
pe), nor should they change the inferred type to `Optional[...]` (despite the rule that does
is for annotated arguments with a default value of `None`). The assumption here is that othe
ode will ensure that the variable is given a value of the proper type, and all uses can assum
at the variable has the given type.

e `# type: ignore` comment should be put on the line that the error refers to:

```
import http.client
errors = {
    'not_found': http.client.NOT_FOUND  # type: ignore
}
```

`# type: ignore` comment on a line by itself at the top of a file, before any docstrings,
nports, or other executable code, silences all errors in the file. Blank lines and other
omments, such as shebang lines and coding cookies, may precede the `# type: ignore`
omment.

some cases, linting tools or other comments may be needed on the same line as a type
omment. In these cases, the type comment should be before other comments and linting
arkers:

> *# type: ignore # <comment or other marker>*

type hinting proves useful in general, a syntax for typing variables may be provided in a
ture Python version. (**UPDATE**: This syntax was added in Python 3.6 through PEP 526.)

asts

ccasionally the type checker may need a different kind of hint: the programmer may know
at an expression is of a more constrained type than a type checker may be able to infer. Fo
ample:

```
from typing import List, cast

def find_first_str(a: List[object]) -> str:
    index = next(i for i, x in enumerate(a) if isinstance(x, str))
    # We only get here if there's at least one string in a
    return cast(str, a[index])
```

›me type checkers may not be able to infer that the type of `a[index]` is `str` and only infer ›ject or `Any`, but we know that (if the code gets to that point) it must be a string. The ›st(t, x) call tells the type checker that we are confident that the type of `x` is `t`. At runtim cast always returns the expression unchanged – it does not check the type, and it does not ›nvert or coerce the value.

ısts differ from type comments (see the previous section). When using a type comment, th pe checker should still verify that the inferred type is consistent with the stated type. Wher ;ing a cast, the type checker should blindly believe the programmer. Also, casts can be use expressions, while type comments only apply to assignments.

ewType helper function

ıere are also situations where a programmer might want to avoid logical errors by creating nple classes. For example:

```
class UserId(int):
    pass

def get_by_user_id(user_id: UserId):
    ...
```

›wever, this approach introduces a runtime overhead. To avoid this, `typing.py` provides a ‹lper function `NewType` that creates simple unique types with almost zero runtime overhead ›r a static type checker `Derived = NewType('Derived', Base)` is roughly equivalent to a ‹finition:

```
class Derived(Base):
    def __init__(self, _x: Base) -> None:
        ...
```

hile at runtime, `NewType('Derived', Base)` returns a dummy function that simply returns its gument. Type checkers require explicit casts from `int` where `UserId` is expected, while ıplicitly casting from `UserId` where `int` is expected. Examples:

```
JserId = NewType('UserId', int)

def name_by_id(user_id: UserId) -> str:
    ...

JserId('user')          # Fails type check

name_by_id(42)          # Fails type check
name_by_id(UserId(42))  # OK

num = UserId(5) + 1     # type: int
```

`ewType` accepts exactly two arguments: a name for the new unique type, and a base class. ie latter should be a proper class (i.e., not a type construct like `Union`, etc.), or another iique type created by calling `NewType`. The function returned by `NewType` accepts only one gument; this is equivalent to supporting only one constructor accepting an instance of the ise class (see above). Example:

```
class PacketId:
    def __init__(self, major: int, minor: int) -> None:
        self._major = major
        self._minor = minor

TcpPacketId = NewType('TcpPacketId', PacketId)

packet = PacketId(100, 100)
tcp_packet = TcpPacketId(packet)  # OK

tcp_packet = TcpPacketId(127, 0)  # Fails in type checker and at runtime
```

oth `isinstance` and `issubclass`, as well as subclassing will fail for `NewType('Derived', Base)` ice function objects don't support these operations.

:ub Files

ub files are files containing type hints that are only for use by the type checker, not at ntime. There are several use cases for stub files:

- Extension modules
- Third-party modules whose authors have not yet added type hints
- Standard library modules for which type hints have not yet been written
- Modules that must be compatible with Python 2 and 3
- Modules that use annotations for other purposes

ub files have the same syntax as regular Python modules. There is one feature of the `typin` odule that is different in stub files: the `@overload` decorator described below.

ıe type checker should only check function signatures in stub files; It is recommended that ınction bodies in stub files just be a single ellipsis (`...`).

ıe type checker should have a configurable search path for stub files. If a stub file is found e type checker should not read the corresponding "real" module.

'hile stub files are syntactically valid Python modules, they use the `.pyi` extension to make ɔssible to maintain stub files in the same directory as the corresponding real module. This so reinforces the notion that no runtime behavior should be expected of stub files.

ɔditional notes on stub files:

- Modules and variables imported into the stub are not considered exported from the stub unless the import uses the `import ... as ...` form or the equivalent `from ... import ... as ...` form. (*UPDATE:* To clarify, the intention here is that only names imported using the form `X as X` will be exported, i.e. the name before and after `as` mu be the same.)
- However, as an exception to the previous bullet, all objects imported into a stub using `from ... import *` are considered exported. (This makes it easier to re-export all object from a given module that may vary by Python version.)
- Just like in normal Python files, submodules automatically become exported attributes of their parent module when imported. For example, if the `spam` package has the following directory structure:

  ```
  spam/
      __init__.pyi
      ham.pyi
  ```

  where `__init__.pyi` contains a line such as `from . import ham` or `from .ham import Ham`, then `ham` is an exported attribute of `spam`.
- Stub files may be incomplete. To make type checkers aware of this, the file can contain the following code:

  ```
  def __getattr__(name) -> Any: ...
  ```

  Any identifier not defined in the stub is therefore assumed to be of type `Any`.

ınction/method overloading

ıe `@overload` decorator allows describing functions and methods that support multiple fferent combinations of argument types. This pattern is used frequently in builtin modules ıd types. For example, the `__getitem__()` method of the `bytes` type can be described as llows:

```
from typing import overload

class bytes:
    ...
    @overload
    def __getitem__(self, i: int) -> int: ...
    @overload
    def __getitem__(self, s: slice) -> bytes: ...
```

ıis description is more precise than would be possible using unions (which cannot express e relationship between the argument and return types):

```
from typing import Union

class bytes:
    ...
    def __getitem__(self, a: Union[int, slice]) -> Union[int, bytes]: ...
```

ıother example where `@overload` comes in handy is the type of the builtin `map()` function, ıich takes a different number of arguments depending on the type of the callable:

```
from typing import Callable, Iterable, Iterator, Tuple, TypeVar, overload

T1 = TypeVar('T1')
T2 = TypeVar('T2')
S = TypeVar('S')

@overload
def map(func: Callable[[T1], S], iter1: Iterable[T1]) -> Iterator[S]: ...
@overload
def map(func: Callable[[T1, T2], S],
        iter1: Iterable[T1], iter2: Iterable[T2]) -> Iterator[S]: ...
# ... and we could add more items to support more than two iterables
```

ɔte that we could also easily add items to support `map(None, ...)`:

```
@overload
def map(func: None, iter1: Iterable[T1]) -> Iterable[T1]: ...
@overload
def map(func: None,
        iter1: Iterable[T1],
        iter2: Iterable[T2]) -> Iterable[Tuple[T1, T2]]: ...
```

ses of the `@overload` decorator as shown above are suitable for stub files. In regular odules, a series of `@overload`-decorated definitions must be followed by exactly one non- `verload`-decorated definition (for the same function/method). The `@overload`-decorated efinitions are for the benefit of the type checker only, since they will be overwritten by the on-`@overload`-decorated definition, while the latter is used at runtime but should be ignore a type checker. At runtime, calling a `@overload`-decorated function directly will raise `tImplementedError`. Here's an example of a non-stub overload that can't easily be expressec ing a union or a type variable:

```
@overload
def utf8(value: None) -> None:
    pass
@overload
def utf8(value: bytes) -> bytes:
    pass
@overload
def utf8(value: unicode) -> bytes:
    pass
def utf8(value):
    <actual implementation>
```

OTE: While it would be possible to provide a multiple dispatch implementation using this ntax, its implementation would require using `sys._getframe()`, which is frowned upon. Also esigning and implementing an efficient multiple dispatch mechanism is hard, which is why evious attempts were abandoned in favor of `functools.singledispatch()`. (See PEP 443, pecially its section "Alternative approaches".) In the future we may come up with a tisfactory multiple dispatch design, but we don't want such a design to be constrained by e overloading syntax defined for type hints in stub files. It is also possible that both feature ll develop independent from each other (since overloading in the type checker has differe e cases and requirements than multiple dispatch at runtime – e.g. the latter is unlikely to pport generic types).

constrained `TypeVar` type can often be used instead of using the `@overload` decorator. For ample, the definitions of `concat1` and `concat2` in this stub file are equivalent:

```
from typing import TypeVar, Text

AnyStr = TypeVar('AnyStr', Text, bytes)

def concat1(x: AnyStr, y: AnyStr) -> AnyStr: ...

@overload
def concat2(x: str, y: str) -> str: ...
@overload
def concat2(x: bytes, y: bytes) -> bytes: ...
```

›me functions, such as `map` or `bytes.__getitem__` above, can't be represented precisely usin‹ pe variables. However, unlike `@overload`, type variables can also be used outside stub files. ‹e recommend that `@overload` is only used in cases where a type variable is not sufficient, ıe to its special stub-only status.

ıother important difference between type variables such as `AnyStr` and using `@overload` is at the prior can also be used to define constraints for generic class type parameters. For ‹ample, the type parameter of the generic class `typing.IO` is constrained (only `IO[str]`, `)[bytes]` and `IO[Any]` are valid):

```
class IO(Generic[AnyStr]): ...
```

oring and distributing stub files

ıe easiest form of stub file storage and distribution is to put them alongside Python odules in the same directory. This makes them easy to find by both programmers and the ols. However, since package maintainers are free not to add type hinting to their packages ird-party stubs installable by `pip` from PyPI are also supported. In this case we have to ›nsider three issues: naming, versioning, installation path.

ıis PEP does not provide a recommendation on a naming scheme that should be used for ird-party stub file packages. Discoverability will hopefully be based on package popularity, ‹e with Django packages for example.

ıird-party stubs have to be versioned using the lowest version of the source package that i ›mpatible. Example: FooPackage has versions 1.0, 1.1, 1.2, 1.3, 2.0, 2.1, 2.2. There are API ıanges in versions 1.1, 2.0 and 2.2. The stub file package maintainer is free to release stubs r all versions but at least 1.0, 1.1, 2.0 and 2.2 are needed to enable the end user type check l versions. This is because the user knows that the closest *lower or equal* version of stubs is ›mpatible. In the provided example, for FooPackage 1.3 the user would choose stubs versic 1.

›te that if the user decides to use the "latest" available source package, using the "latest" ub files should generally also work if they're updated often.

ıird-party stub packages can use any location for stub storage. Type checkers should searc r them using PYTHONPATH. A default fallback directory that is always checked is `ıared/typehints/pythonX.Y/` (for some PythonX.Y as determined by the type checker, not just e installed version). Since there can only be one package installed for a given Python ersion per environment, no additional versioning is performed under that directory (just like are directory installs by `pip` in site-packages). Stub file package authors might use the llowing snippet in `setup.py`:

```
...
data_files=[
    (
        'shared/typehints/python{}.{}'.format(*sys.version_info[:2]),
        pathlib.Path(SRC_PATH).glob('**/*.pyi'),
    ),
],
...
```

PDATE: As of June 2018 the recommended way to distribute type hints for third-party ackages has changed – in addition to typeshed (see the next section) there is now a andard for distributing type hints, PEP 561. It supports separately installable packages ontaining stubs, stub files included in the same distribution as the executable code of a ackage, and inline type hints, the latter two options enabled by including a file named `.typed` in the package.)

ne Typeshed Repo

nere is a shared repository where useful stubs are being collected. Policies regarding the ubs collected here will be decided separately and reported in the repo's documentation. ote that stubs for a given package will not be included here if the package owners have ecifically requested that they be omitted.

xceptions

o syntax for listing explicitly raised exceptions is proposed. Currently the only known use se for this feature is documentational, in which case the recommendation is to put this formation in a docstring.

ne `typing` Module

open the usage of static type checking to Python 3.5 as well as older versions, a uniform amespace is required. For this purpose, a new module in the standard library is introduced lled `typing`.

defines the fundamental building blocks for constructing types (e.g. `Any`), types presenting generic variants of builtin collections (e.g. `List`), types representing generic ollection ABCs (e.g. `Sequence`), and a small collection of convenience definitions.

ote that special type constructs, such as `Any`, `Union`, and type variables defined using `peVar` are only supported in the type annotation context, and `Generic` may only be used as base class. All of these (except for unparameterized generics) will raise `TypeError` if appear `isinstance` or `issubclass`.

ndamental building blocks:

- Any, used as `def get(key: str) -> Any: ...`
- Union, used as `Union[Type1, Type2, Type3]`
- Callable, used as `Callable[[Arg1Type, Arg2Type], ReturnType]`
- Tuple, used by listing the element types, for example `Tuple[int, int, str]`. The empty tuple can be typed as `Tuple[()]`. Arbitrary-length homogeneous tuples can be expressed using one type and ellipsis, for example `Tuple[int, ...]`. (The `...` here are part of the syntax, a literal ellipsis.)
- TypeVar, used as `X = TypeVar('X', Type1, Type2, Type3)` or simply `Y = TypeVar('Y')` (see above for more details)
- Generic, used to create user-defined generic classes
- Type, used to annotate class objects

eneric variants of builtin collections:

- Dict, used as `Dict[key_type, value_type]`
- DefaultDict, used as `DefaultDict[key_type, value_type]`, a generic variant of `collections.defaultdict`
- List, used as `List[element_type]`
- Set, used as `Set[element_type]`. See remark for `AbstractSet` below.
- FrozenSet, used as `FrozenSet[element_type]`

ote: `Dict`, `DefaultDict`, `List`, `Set` and `FrozenSet` are mainly useful for annotating return lues. For arguments, prefer the abstract collection types defined below, e.g. `Mapping`, `equence` or `AbstractSet`.

eneric variants of container ABCs (and a few non-containers):

- Awaitable
- AsyncIterable
- AsyncIterator
- ByteString
- Callable (see above, listed here for completeness)
- Collection

- Container
- ContextManager
- Coroutine
- Generator, used as `Generator[yield_type, send_type, return_type]`. This represents the return value of generator functions. It is a subtype of `Iterable` and it has additional typ variables for the type accepted by the `send()` method (it is contravariant in this variabl – a generator that accepts sending it `Employee` instance is valid in a context where a generator is required that accepts sending it `Manager` instances) and the return type of the generator.
- Hashable (not generic, but present for completeness)
- ItemsView
- Iterable
- Iterator
- KeysView
- Mapping
- MappingView
- MutableMapping
- MutableSequence
- MutableSet
- Sequence
- Set, renamed to `AbstractSet`. This name change was required because `Set` in the `typin` module means `set()` with generics.
- Sized (not generic, but present for completeness)
- ValuesView

few one-off types are defined that test for single special methods (similar to `Hashable` or `zed`):

- Reversible, to test for `__reversed__`
- SupportsAbs, to test for `__abs__`
- SupportsComplex, to test for `__complex__`
- SupportsFloat, to test for `__float__`
- SupportsInt, to test for `__int__`
- SupportsRound, to test for `__round__`
- SupportsBytes, to test for `__bytes__`

onvenience definitions:

- Optional, defined by `Optional[t] == Union[t, None]`

- Text, a simple alias for `str` in Python 3, for `unicode` in Python 2
- AnyStr, defined as `TypeVar('AnyStr', Text, bytes)`
- NamedTuple, used as `NamedTuple(type_name, [(field_name, field_type), ...])` and equivalent to `collections.namedtuple(type_name, [field_name, ...])`. This is useful to declare the types of the fields of a named tuple type.
- NewType, used to create unique types with little runtime overhead `UserId = NewType('UserId', int)`
- cast(), described earlier
- @no_type_check, a decorator to disable type checking per class or function (see below
- @no_type_check_decorator, a decorator to create your own decorators with the same meaning as `@no_type_check` (see below)
- @type_check_only, a decorator only available during type checking for use in stub files (see above); marks a class or function as unavailable during runtime
- @overload, described earlier
- get_type_hints(), a utility function to retrieve the type hints from a function or method. Given a function or method object, it returns a dict with the same format as `__annotations__`, but evaluating forward references (which are given as string literals) as expressions in the context of the original function or method definition.
- TYPE_CHECKING, `False` at runtime but `True` to type checkers

O related types:

- IO (generic over `AnyStr`)
- BinaryIO (a simple subtype of `IO[bytes]`)
- TextIO (a simple subtype of `IO[str]`)

pes related to regular expressions and the `re` module:

- Match and Pattern, types of `re.match()` and `re.compile()` results (generic over `AnyStr`)

ıggested syntax for Python 2.7 and straddling code

ɔme tools may want to support type annotations in code that must be compatible with thon 2.7. For this purpose this PEP has a suggested (but not mandatory) extension where nction annotations are placed in a `# type:` comment. Such a comment must be placed ımediately following the function header (before the docstring). An example: the following thon 3 code:

```
def embezzle(self, account: str, funds: int = 1000000, *fake_receipts: str) -> None
    """Embezzle funds from account using fake receipts."""
    <code goes here>
```

equivalent to the following:

```
def embezzle(self, account, funds=1000000, *fake_receipts):
    # type: (str, int, *str) -> None
    """Embezzle funds from account using fake receipts."""
    <code goes here>
```

ote that for methods, no type is needed for `self`.

or an argument-less method it would look like this:

```
def load_cache(self):
    # type: () -> bool
    <code>
```

ometimes you want to specify the return type for a function or method without (yet) ecifying the argument types. To support this explicitly, the argument list may be replaced th an ellipsis. Example:

```
def send_email(address, sender, cc, bcc, subject, body):
    # type: (...) -> bool
    """Send an email message.  Return True if successful."""
    <code>
```

ometimes you have a long list of parameters and specifying their types in a single `# type:` omment would be awkward. To this end you may list the arguments one per line and add a `type:` comment per line after an argument's associated comma, if any. To specify the retur pe use the ellipsis syntax. Specifying the return type is not mandatory and not every gument needs to be given a type. A line with a `# type:` comment should contain exactly ne argument. The type comment for the last argument (if any) should precede the close arenthesis. Example:

```
def send_email(address,     # type: Union[str, List[str]]
               sender,      # type: str
               cc,          # type: Optional[List[str]]
               bcc,         # type: Optional[List[str]]
               subject='',
               body=None    # type: List[str]
               ):
    # type: (...) -> bool
    """Send an email message.  Return True if successful."""
    <code>
```

otes:

- Tools that support this syntax should support it regardless of the Python version being checked. This is necessary in order to support code that straddles Python 2 and Pythor 3.
- It is not allowed for an argument or return value to have both a type annotation and a type comment.
- When using the short form (e.g. `# type: (str, int) -> None`) every argument must be accounted for, except the first argument of instance and class methods (those are usually omitted, but it's allowed to include them).
- The return type is mandatory for the short form. If in Python 3 you would omit some argument or the return type, the Python 2 notation should use `Any`.
- When using the short form, for `*args` and `**kwds`, put 1 or 2 stars in front of the corresponding type annotation. (As with Python 3 annotations, the annotation here denotes the type of the individual argument values, not of the tuple/dict that you receive as the special argument value `args` or `kwds`.)
- Like other type comments, any names used in the annotations must be imported or defined by the module containing the annotation.
- When using the short form, the entire annotation must be one line.
- The short form may also occur on the same line as the close parenthesis, e.g.:

  ```
  def add(a, b):  # type: (int, int) -> int
      return a + b
  ```

- Misplaced type comments will be flagged as errors by a type checker. If necessary, suc comments could be commented twice. For example:

  ```
  def f():
      '''Docstring'''
      # type: () -> None  # Error!

  def g():
      '''Docstring'''
      # # type: () -> None  # This is OK
  ```

'hen checking Python 2.7 code, type checkers should treat the `int` and `long` types as ¡uivalent. For parameters typed as `Text`, arguments of type `str` as well as `unicode` should b :ceptable.

ejected Alternatives

ıring discussion of earlier drafts of this PEP, various objections were raised and alternatives ere proposed. We discuss some of these here and explain why we reject them.

everal main objections were raised.

'hich brackets for generic type parameters?

ost people are familiar with the use of angular brackets (e.g. `List<int>`) in languages like ++, Java, C# and Swift to express the parameterization of generic types. The problem with ese is that they are really hard to parse, especially for a simple-minded parser like Python. most languages the ambiguities are usually dealt with by only allowing angular brackets in ecific syntactic positions, where general expressions aren't allowed. (And also by using ver owerful parsing techniques that can backtrack over an arbitrary section of code.)

ıt in Python, we'd like type expressions to be (syntactically) the same as other expressions, that we can use e.g. variable assignment to create type aliases. Consider this simple type :pression:

```
List<int>
```

om the Python parser's perspective, the expression begins with the same four tokens IAME, LESS, NAME, GREATER) as a chained comparison:

```
a < b > c  # I.e., (a < b) and (b > c)
```

'e can even make up an example that could be parsed both ways:

```
a < b > [ c ]
```

ssuming we had angular brackets in the language, this could be interpreted as either of the llowing two:

```
(a<b>)[c]      # I.e., (a<b>).__getitem__(c)
a < b > ([c])  # I.e., (a < b) and (b > [c])
```

would surely be possible to come up with a rule to disambiguate such cases, but to most ers the rules would feel arbitrary and complex. It would also require us to dramatically ange the CPython parser (and every other parser for Python). It should be noted that thon's current parser is intentionally "dumb" – a simple grammar is easier for users to ason about.

r all these reasons, square brackets (e.g. `List[int]`) are (and have long been) the preferrec ntax for generic type parameters. They can be implemented by defining the `__getitem__()` ethod on the metaclass, and no new syntax is required at all. This option works in all recen rsions of Python (starting with Python 2.2). Python is not alone in this syntactic choice – neric classes in Scala also use square brackets.

hat about existing uses of annotations?

ne line of argument points out that PEP 3107 explicitly supports the use of arbitrary pressions in function annotations. The new proposal is then considered incompatible with e specification of PEP 3107.

ur response to this is that, first of all, the current proposal does not introduce any direct compatibilities, so programs using annotations in Python 3.4 will still work correctly and thout prejudice in Python 3.5.

e do hope that type hints will eventually become the sole use for annotations, but this will quire additional discussion and a deprecation period after the initial roll-out of the typing odule with Python 3.5. The current PEP will have provisional status (see PEP 411) until thon 3.6 is released. The fastest conceivable scheme would introduce silent deprecation of n-type-hint annotations in 3.6, full deprecation in 3.7, and declare type hints as the only lowed use of annotations in Python 3.8. This should give authors of packages that use notations plenty of time to devise another approach, even if type hints become an ernight success.

PDATE: As of fall 2017, the timeline for the end of provisional status for this PEP and for th `ping.py` module has changed, and so has the deprecation schedule for other uses of notations. For the updated schedule see PEP 563.)

other possible outcome would be that type hints will eventually become the default eaning for annotations, but that there will always remain an option to disable them. For th rpose the current proposal defines a decorator `@no_type_check` which disables the default terpretation of annotations as type hints in a given class or function. It also defines a meta ecorator `@no_type_check_decorator` which can be used to decorate a decorator (!), causing nnotations in any function or class decorated with the latter to be ignored by the type hecker.

here are also `# type: ignore` comments, and static checkers should support configuration ptions to disable type checking in selected packages.

espite all these options, proposals have been circulated to allow type hints and other forms annotations to coexist for individual arguments. One proposal suggests that if an nnotation for a given argument is a dictionary literal, each key represents a different form c nnotation, and the key `'type'` would be use for type hints. The problem with this idea and variants is that the notation becomes very "noisy" and hard to read. Also, in most cases here existing libraries use annotations, there would be little need to combine them with pe hints. So the simpler approach of selectively disabling type hints appears sufficient.

he problem of forward declarations

he current proposal is admittedly sub-optimal when type hints must contain forward ferences. Python requires all names to be defined by the time they are used. Apart from rcular imports this is rarely a problem: "use" here means "look up at runtime", and with mo orward" references there is no problem in ensuring that a name is defined before the nction using it is called.

he problem with type hints is that annotations (per PEP 3107, and similar to default values) e evaluated at the time a function is defined, and thus any names used in an annotation ust be already defined when the function is being defined. A common scenario is a class efinition whose methods need to reference the class itself in their annotations. (More eneral, it can also occur with mutually recursive classes.) This is natural for container types, r example:

```
class Node:
    """Binary tree node."""

    def __init__(self, left: Node, right: Node):
        self.left = left
        self.right = right
```

s written this will not work, because of the peculiarity in Python that class names become efined once the entire body of the class has been executed. Our solution, which isn't articularly elegant, but gets the job done, is to allow using string literals in annotations. ost of the time you won't have to use this though – most *uses* of type hints are expected t ference builtin types or types defined in other modules.

counterproposal would change the semantics of type hints so they aren't evaluated at ntime at all (after all, type checking happens off-line, so why would type hints need to be 'aluated at runtime at all). This of course would run afoul of backwards compatibility, since e Python interpreter doesn't actually know whether a particular annotation is meant to be pe hint or something else.

compromise is possible where a `__future__` import could enable turning *all* annotations in ven module into string literals, as follows:

```
from __future__ import annotations

class ImSet:
    def add(self, a: ImSet) -> List[ImSet]: ...

assert ImSet.add.__annotations__ == {'a': 'ImSet', 'return': 'List[ImSet]'}
```

ıch a `__future__` import statement may be proposed in a separate PEP.

'PDATE: That `__future__` import statement and its consequences are discussed in PEP 563.)

ıe double colon

few creative souls have tried to invent solutions for this problem. For example, it was 'oposed to use a double colon (`::`) for type hints, solving two problems at once: sambiguating between type hints and other annotations, and changing the semantics to 'eclude runtime evaluation. There are several things wrong with this idea, however.

- It's ugly. The single colon in Python has many uses, and all of them look familiar because they resemble the use of the colon in English text. This is a general rule of thumb by which Python abides for most forms of punctuation; the exceptions are typically well known from other programming languages. But this use of `::` is unheard of in English, and in other languages (e.g. C++) it is used as a scoping operator, which a very different beast. In contrast, the single colon for type hints reads naturally – and no wonder, since it was carefully designed for this purpose (the idea long predates PEI 3107). It is also used in the same fashion in other languages from Pascal to Swift.
- What would you do for return type annotations?
- It's actually a feature that type hints are evaluated at runtime.
    - Making type hints available at runtime allows runtime type checkers to be built o top of type hints.
    - It catches mistakes even when the type checker is not run. Since it is a separate program, users may choose not to run it (or even install it), but might still want tc

use type hints as a concise form of documentation. Broken type hints are no use even for documentation.

- Because it's new syntax, using the double colon for type hints would limit them to cod that works with Python 3.5 only. By using existing syntax, the current proposal can easi work for older versions of Python 3. (And in fact mypy supports Python 3.2 and newer.
- If type hints become successful we may well decide to add new syntax in the future to declare the type for variables, for example `var age: int = 42`. If we were to use a doubl colon for argument type hints, for consistency we'd have to use the same convention for future syntax, perpetuating the ugliness.

ther forms of new syntax

few other forms of alternative syntax have been proposed, e.g. the introduction of a `where` yword, and Cobra-inspired `requires` clauses. But these all share a problem with the doubl lon: they won't work for earlier versions of Python 3. The same would apply to a new `_future__` import.

ther backwards compatible conventions

e ideas put forward include:

- A decorator, e.g. `@typehints(name=str, returns=str)`. This could work, but it's pretty verbose (an extra line, and the argument names must be repeated), and a far cry in elegance from the PEP 3107 notation.
- Stub files. We do want stub files, but they are primarily useful for adding type hints to existing code that doesn't lend itself to adding type hints, e.g. 3rd party packages, cod that needs to support both Python 2 and Python 3, and especially extension modules. For most situations, having the annotations in line with the function definitions makes them much more useful.
- Docstrings. There is an existing convention for docstrings, based on the Sphinx notatio (`:type arg1: description`). This is pretty verbose (an extra line per parameter), and not very elegant. We could also make up something new, but the annotation syntax is harc to beat (because it was designed for this very purpose).

s also been proposed to simply wait another release. But what problem would that solve? ould just be procrastination.

EP Development Process

live draft for this PEP lives on GitHub. There is also an issue tracker, where much of the chnical discussion takes place.

e draft on GitHub is updated regularly in small increments. The official PEPS repo is sually) only updated when a new draft is posted to python-dev.

cknowledgements

is document could not be completed without valuable input, encouragement and advice om Jim Baker, Jeremy Siek, Michael Matson Vitousek, Andrey Vlasovskikh, Radomir pieralski, Peter Ludemann, and the BDFL-Delegate, Mark Shannon.

fluences include existing languages, libraries and frameworks mentioned in PEP 482. Many anks to their creators, in alphabetical order: Stefan Behnel, William Edwards, Greg Ewing, rry Hastings, Anders Hejlsberg, Alok Menghrajani, Travis E. Oliphant, Joe Pamer, Raoul-briel Urma, and Julien Verlaguet.

opyright

---

urce: https://github.com/python/peps/blob/main/peps/pep-0484.rst

st modified: 2023-09-09 17:39:29 GMT