PEP 484 -- Type Hints

PEP:	484
Title:	Type Hints
Author:	Guido van Rossum <guido at python.org>, Jukka Lehtosalo <jukka.lehtosalo at iki.fi>, Łukasz Langa <lukasz at python.org>
BDFL-Delegate:	Mark Shannon
Discussions-To:	Python-Dev <python-dev at python.org (mailto:python-dev@python.org?subject=PEP%20484)>
Status:	Provisional
Type:	Standards Track
Created:	29-Sep-2014
Python-Version:	3.5
Post-History:	16-Jan-2015,20-Mar-2015,17-Apr-2015,20-May-2015,22-May-2015
Resolution:	https://mail.python.org/pipermail/python-dev/2015-May/140104.html (https://mail.python.org/pipermail/python-dev/2015-May/140104.html)

Contents

Abstract (#id10)

PEP 3107 (/dev/peps/pep-3107) introduced syntax for function annotations, but the semantics were deliberately left undefined. There has now been enough 3rd party usage for static type analysis that the community would benefit from a standard vocabulary and baseline tools within the standard library.

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

Note that this PEP still explicitly does NOT prevent other uses of annotations, nor does it require (or forbid) any particular processing of annotations, even when they conform to this specification. It simply enables better coordination, as PEP 333 (/dev/peps/pep-0333) did for web frameworks.

For example, here is a simple function whose argument and return type are declared in the annotations:

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

While these annotations are available at runtime through the usual `__annotations__` attribute, *no type checking happens at runtime*. Instead, the proposal assumes the existence of a separate off-line type checker which users can run over their source code voluntarily. Essentially, such a type checker acts as a very powerful linter. (While it would of course be possible for individual users to employ a similar checker at run time for Design By Contract enforcement or JIT optimization, those tools are not yet as mature.)

The proposal is strongly inspired by mypy [mypy] (#mypy). For example, the type "sequence of integers" can be written as `Sequence[int]`. The square brackets mean that no new syntax needs to be added to the language. The example here uses a custom type `Sequence`, imported from a pure-Python 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).

The type system supports unions, generic types, and a special type named Any which is consistent with (i.e. assignable to and from) all types. This latter feature is taken from the idea of gradual typing. Gradual typing and the full type system are explained in PEP 483 (/dev/peps/pep-0483).

Other approaches from which we have borrowed or to which ours can be compared and contrasted are described in PEP 482 (/dev/peps/pep-0482).

Rationale and Goals (#id11)

PEP 3107 (/dev/peps/pep-3107) added support for arbitrary annotations on parts of a function definition. Although no meaning was assigned to annotations then, there has always been an implicit goal to use them for type hinting [gvr-artima] (#gvr-artima), which is listed as the first possible use case in said PEP.

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

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

Non-goals (#id12)

While the proposed typing module will contain some building blocks for runtime type checking -- in particular the `get_type_hints()` function -- third party packages would have to be developed to implement specific runtime type checking functionality, for example using decorators or metaclasses. Using type hints for performance optimizations is left as an exercise for the reader.

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

The meaning of annotations (#id13)

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

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

(Note that the return type of `__init__` ought to be annotated with `-> None`. The reason for this is subtle. If `__init__` assumed a return annotation of `-> None`, would that mean that an argument-less, un-annotated `__init__` method should still be type-checked? Rather than leaving 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 other methods.)

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

Type checkers are expected to attempt to infer as much information as necessary. The minimum requirement is to handle the builtin decorators `@property`, `@staticmethod` and `@classmethod`.

Type Definition Syntax (#id14)

The syntax leverages PEP 3107 (/dev/peps/pep-3107)-style annotations with a number of extensions described in sections below. In its basic form, type hinting is used by filling function annotation slots with classes:

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

This states that the expected type of the name argument is `str`. Analogically, the expected return type is `str`.

Expressions whose type is a subtype of a specific argument type are also accepted for that argument.

Acceptable type hints (#id15)

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

While annotations are normally the best format for type hints, there are times when it is more appropriate to represent them by a special comment, or in a separately distributed stub file. (See below for examples.)

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

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

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

All newly introduced names used to support features described in following sections (such as Any and Union) are available in the `typing` module.

Using None (#id16)

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

Type aliases (#id17)

Type aliases are defined by simple variable assignments:

```
Url = str


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

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

Type aliases may be as complex as type hints in annotations -- anything that is acceptable as a type 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]
```

This 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]]
```

Callable (#id18)

Frameworks expecting callback functions of specific signatures might be type hinted using `Callable[[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
```

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

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

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

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

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

Generics (#id19)

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. Example:

```
from typing import Mapping, Set

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

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

```
from typing import Sequence, TypeVar

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

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

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

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

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

The function `concat` can be called with either two `str` arguments or two `bytes` arguments, but not with a mix of `str` and `bytes` arguments.

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

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

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

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

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

Additionally, 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)
```

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

User-defined generic types (#id20)

You 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
    ...
```

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

```
from typing import TypeVar, Iterator

T = TypeVar('T')

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

That class definition is equivalent to:

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

You 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]):
    ...
```

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]
    ...
```

Generic metaclasses are not supported.

Scoping rules for type variables (#id21)

Type variables follow normal name resolution rules. However, there are some special cases in the static typechecking context:

- A type variable used in a generic function could be inferred to represent different types 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
```

Instantiating generic classes and type erasure (#id22)

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

```
from typing import TypeVar, Generic

T = TypeVar('T')

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

To create Node instances you call `Node()` just as for a regular class. At runtime the type (class) of the instance will be Node. But what type does it have to the type checker? The answer depends 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 the 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]
```

In case the inferred type uses `[Any]` but the intended type is more specific, you can use a type comment (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]
```

Alternatively, 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
```

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

Using generic classes (parameterized or not) to access attributes will result in type check failure. Outside the class definition body, a class attribute cannot be assigned, and can only be looked up by accessing it through a class instance that does not have an instance attribute with 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
```

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

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

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

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

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

Arbitrary generic types as base classes (#id23)

`Generic[T]` is only valid as a base class -- it's not a proper type. However, user-defined generic types such as `LinkedList[T]` from the above example and built-in generic types and ABCs such as `List[T]` and `Iterable[T]` are valid both as types and as base classes. For example, we can 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
```

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

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

```
from typing import TypeVar, Iterable, Container

T = TypeVar('T')

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

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

Also consider the following example:

```
from typing import TypeVar, Mapping

T = TypeVar('T')

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

In this case MyDict has a single parameter, T.

Abstract generic types (#id24)

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.

Type variables with an upper bound (#id25)

A type variable may specify an upper bound using `bound=<type>` (note: <type> itself cannot be parametrized by type variables). This means that an actual type substituted (explicitly or implicitly) for the type variable must be a subtype of the boundary type. A common example is the definition of a `Comparable` type that works well enough to catch the most common errors:

```
from typing import TypeVar

class Comparable(metaclass=ABCMeta):
    @abstractmethod
    def __lt__(self, other: Any) -> bool: ...
    ...  # __gt__ etc. as well

CT = TypeVar('CT', bound=Comparable)

def min(x: CT, y: CT) -> CT:
    if x < y:
        return x
    else:
        return y

min(1, 2)      # ok, return type int
min('x', 'y')  # ok, return type str
```

(Note that this is not ideal -- for example `min('x', 1)` is invalid at runtime but a type checker would simply infer the return type `Comparable`. Unfortunately, addressing this would require introducing a much more powerful and also much more complicated concept, F-bounded polymorphism. We may revisit this in the future.)

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

Covariance and contravariance (#id26)

Consider a class `Employee` with a subclass `Manager`. Now suppose we have a function with an argument annotated with `List[Employee]`. Should we be allowed to call this function with a variable of type `List[Manager]` as its argument? Many people would answer "yes, of course" without even considering the consequences. But unless we know more about the function, a type checker should reject such a call: the function might append an `Employee` instance to the list, which would violate the variable's type in the caller.

It turns out such an argument acts *contravariantly*, whereas the intuitive answer (which is correct in case the function doesn't mutate its argument!) requires the argument to act *covariantly*. A longer introduction to these concepts can be found on Wikipedia [wiki-variance] (#wiki-variance) and in PEP 483 (/dev/peps/pep-0483); here we just show how to control a type checker's behavior.

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

To facilitate the declaration of container types where covariant or contravariant type checking is acceptable, type variables accept keyword arguments `covariant=True` or `contravariant=True`. At most one of these may be passed. Generic types defined with such variables are considered covariant or contravariant in the corresponding variable. By convention, it is recommended to use names ending in `_co` for type variables defined with `covariant=True` and names ending in `_contra` for that defined with `contravariant=True`.

A 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
```

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

Note: Covariance or contravariance is *not* a property of a type variable, but a property of a generic class defined using this variable. Variance is only applicable to generic types; generic functions do not have this property. The latter should be defined using only type variables without `covariant` or `contravariant` keyword arguments. For example, the following example is 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
```

while 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
    ...
```

The numeric tower (#id27)

PEP 3141 (/dev/peps/pep-3141) defines Python's numeric tower, and the stdlib module `numbers` implements the corresponding ABCs (`Number`, `Complex`, `Real`, `Rational` and `Integral`). There are some issues with these ABCs, but the built-in concrete numeric classes `complex`, `float` and `int` are ubiquitous (especially the latter two :-).

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

Forward references (#id28)

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

A situation where this occurs commonly is the definition of a container class, where the class being defined occurs in the signature of some of the methods. For example, the following code (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
```

To address this, we write:

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

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

Moreover, the expression should be parseable as a valid type hint, i.e., it is constrained by the rules from the section Acceptable type hints (#acceptable-type-hints) above.

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

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

A common use for forward references is when e.g. Django models are needed in the signatures. Typically, each model is in a separate file, and has methods taking arguments whose type involves other models. Because of the way circular imports work in Python, it is often 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
```

Assuming main is imported first, this will fail with an ImportError at the line `from models.a import A` in models/b.py, which is being imported from models/a.py before a has defined class A. The solution is to switch to module-only imports and reference the models by their _module_._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
```

Union types (#id29)

Since accepting a small, limited set of expected types for a single argument is common, there is 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]
    ...
```

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

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

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

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

```
from typing import Optional


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

A past version of this PEP allowed type checkers to assume an optional type when the default value is None, as in this code:

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

This would have been treated as equivalent to:

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

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

Support for singleton types in unions (#id30)

A singleton instance is frequently used to mark some special condition, in particular in situations 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
```

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

```
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
```

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

The Any type (#id31)

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

A function parameter without an annotation is assumed to be annotated with `Any`. If a generic type 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]
    ...
```

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

The `NoReturn` type (#id32)

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

```
from typing import NoReturn

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

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

```
import sys
from typing import NoReturn

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

The checkers will also recognize that the code after calls to such functions is unreachable and will 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
```

The `NoReturn` type is only valid as a return annotation of functions, and considered an error if it 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]:
    ...
```

The type of class objects (#id33)

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

For example, suppose we have the following classes:

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

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

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

Without `Type[]` the best we could do to annotate `new_user()` would be:

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

However 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:
    ...
```

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

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

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

Note 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)
    ...
```

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

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

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

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

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

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

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

Regarding the behavior of `Type[Any]` (or `Type` or `type`), accessing attributes of a variable with this 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`.

`Type` 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
    ...
```

Annotating instance and class methods (#id34)

In most cases the first argument of class and instance methods does not need to be annotated, and it is assumed to have the type of the containing class for instance methods, and a type object type corresponding to the containing class object for class methods. In addition, the first argument in an instance method can be annotated with a type variable. In this case the return type may use the same type variable, thus making that method a generic function. 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
```

The 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
```

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

Version and platform checking (#id35)

Type 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
```

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

Runtime or type checking? (#id36)

Sometimes there's code that must be seen by a type checker (or other static analysis tools) but should not be executed. For such situations the `typing` module defines a constant, `TYPE_CHECKING`, that is considered `True` during type checking (or other static analysis) but `False` at 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
    ...
```

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

This approach may also be useful to handle import cycles.

Arbitrary argument lists and default argument values (#id37)

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

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

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

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

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

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

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

What should the default value look like? Any of the options `""`, `b""` or `None` fails to satisfy the type constraint.

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

Positional-only arguments (#id38)

Some functions are designed to take their arguments only positionally, and expect their callers never to use the argument's name to provide that argument by keyword. All arguments with names beginning with __ are assumed to be positional-only, except if their names 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.
```

Annotating generator functions and coroutines (#id39)

The return type of generator functions can be annotated by the generic type `Generator[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'
```

Coroutines introduced in PEP 492 (/dev/peps/pep-0492) are annotated with the same syntax as ordinary functions. However, the return type annotation corresponds to the type of `await` expression, not to the coroutine type:

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

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

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

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

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

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

Compatibility with other uses of function annotations (#id40)

A number of existing or potential use cases for function annotations exist, which are incompatible with type hinting. These may confuse a static type checker. However, since type hinting annotations have no runtime behavior (other than evaluation of the annotation expression and storing annotations in the `__annotations__` attribute of the function object), this does not make the program incorrect -- it just may cause a type checker to emit spurious warnings or errors.

To mark portions of the program that should not be covered by type hinting, you can use one 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`.

For more details see later sections.

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

Type comments (#id41)

No first-class syntax support for explicitly marking variables as being of a specific type is added by this PEP. To help with type inference in complex cases, a comment of the following format 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]
```

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

Examples 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
    ...
```

In stubs it may be useful to declare the existence of a variable without giving it an initial value. This can be done using PEP 526 (/dev/peps/pep-0526) variable annotation syntax:

```
from typing import IO

stream: IO[str]
```

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

```
from typing import IO

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

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

The `# 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
}
```

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

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

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

If type hinting proves useful in general, a syntax for typing variables may be provided in a future Python version. (**UPDATE**: This syntax was added in Python 3.6 through PEP 526 (/dev/peps/pep-0526).)

Casts (#id42)

Occasionally the type checker may need a different kind of hint: the programmer may know that an expression is of a more constrained type than a type checker may be able to infer. For example:

```
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])
```

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

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

NewType helper function (#id43)

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

```
class UserId(int):
    pass

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

However, this approach introduces a runtime overhead. To avoid this, `typing.py` provides a helper function `NewType` that creates simple unique types with almost zero runtime overhead. For a static type checker `Derived = NewType('Derived', Base)` is roughly equivalent to a definition:

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

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

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

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

UserId('user')          # Fails type check

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

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

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

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

Stub Files (#id44)

Stub files are files containing type hints that are only for use by the type checker, not at runtime. 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

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

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

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

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

Additional 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.

- 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 objects from a given module that may vary by Python version.)

- Just like in normal Python files [importdocs] (#importdocs), 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.

Function/method overloading (#id45)

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

```
from typing import overload

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

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

```
from typing import Union

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

Another example where `@overload` comes in handy is the type of the builtin `map()` function, which 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
```

Note 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]]: ...
```

Uses of the `@overload` decorator as shown above are suitable for stub files. In regular modules, a series of `@overload`-decorated definitions must be followed by exactly one non-`@overload`-decorated definition (for the same function/method). The `@overload`-decorated definitions are for the benefit of the type checker only, since they will be overwritten by the non-`@overload`-decorated definition, while the latter is used at runtime but should be ignored by a type checker. At runtime, calling a `@overload`-decorated function directly will raise `NotImplementedError`. Here's an example of a non-stub overload that can't easily be expressed using 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>
```

NOTE: While it would be possible to provide a multiple dispatch implementation using this syntax, its implementation would require using `sys._getframe()`, which is frowned upon. Also, designing and implementing an efficient multiple dispatch mechanism is hard, which is why previous attempts were abandoned in favor of `functools.singledispatch()`. (See PEP 443 (/dev/peps/pep-0443), especially its section "Alternative approaches".) In the future we may come up with a satisfactory multiple dispatch design, but we don't want such a design to be constrained by the overloading syntax defined for type hints in stub files. It is also possible that both features will develop independent from each other (since overloading in the type checker has different use cases and requirements than multiple dispatch at runtime -- e.g. the latter is unlikely to support generic types).

A constrained `TypeVar` type can often be used instead of using the `@overload` decorator. For example, 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: ...
```

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

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

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

Storing and distributing stub files (#id46)

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

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

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

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

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

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

(*UPDATE:* As of June 2018 the recommended way to distribute type hints for third-party packages has changed -- in addition to typeshed (see the next section) there is now a standard for distributing type hints, PEP 561 (/dev/peps/pep-0561). It supports separately installable packages containing stubs, stub files included in the same distribution as the executable code of a package, and inline type hints, the latter two options enabled by including a file named `py.typed` in the package.)

The Typeshed Repo (#id47)

There is a shared repository where useful stubs are being collected [typeshed] (#typeshed). Policies regarding the stubs collected here will be decided separately and reported in the repo's documentation. Note that stubs for a given package will not be included here if the package owners have specifically requested that they be omitted.

Exceptions (#id48)

No syntax for listing explicitly raised exceptions is proposed. Currently the only known use case for this feature is documentational, in which case the recommendation is to put this information in a docstring.

The `typing` Module (#id49)

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

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

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

Fundamental 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

Generic 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]`

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

Generic 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 type variables for the type accepted by the `send()` method (it is contravariant in this variable -- 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 `typing` module means `set()` with generics.
- Sized (not generic, but present for completeness)
- ValuesView

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

- 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__`

Convenience 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

I/O related types:

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

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

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

Suggested syntax for Python 2.7 and straddling code (#id50)

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

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

is 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>
```

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

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

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

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

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

Sometimes you have a long list of parameters and specifying their types in a single `# type:` comment 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 return type use the ellipsis syntax. Specifying the return type is not mandatory and not every argument needs to be given a type. A line with a `# type:` comment should contain exactly one argument. The type comment for the last argument (if any) should precede the close parenthesis. 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>
```

Notes:

- 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 Python 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, such comments could be commented twice. For example:

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

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

When checking Python 2.7 code, type checkers should treat the `int` and `long` types as equivalent. For parameters typed as `Text`, arguments of type `str` as well as `unicode` should be acceptable.

Rejected Alternatives (#id51)

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

Several main objections were raised.

Which brackets for generic type parameters? (#id52)

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

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

```
List<int>
```

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

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

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

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

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

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

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

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

What about existing uses of annotations? (#id53)

One line of argument points out that PEP 3107 (/dev/peps/pep-3107) explicitly supports the use of arbitrary expressions in function annotations. The new proposal is then considered incompatible with the specification of PEP 3107 (/dev/peps/pep-3107).

Our response to this is that, first of all, the current proposal does not introduce any direct incompatibilities, so programs using annotations in Python 3.4 will still work correctly and without prejudice in Python 3.5.

We do hope that type hints will eventually become the sole use for annotations, but this will require additional discussion and a deprecation period after the initial roll-out of the typing module with Python 3.5. The current PEP will have provisional status (see PEP 411 (/dev/peps/pep-0411)) until Python 3.6 is released. The fastest conceivable scheme would introduce silent deprecation of non-type-hint annotations in 3.6, full deprecation in 3.7, and declare type hints as the only allowed use of annotations in Python 3.8. This should give authors of packages that use annotations plenty of time to devise another approach, even if type hints become an overnight success.

(*UPDATE:* As of fall 2017, the timeline for the end of provisional status for this PEP and for the `typing.py` module has changed, and so has the deprecation schedule for other uses of annotations. For the updated schedule see PEP 563 (/dev/peps/pep-0563).)

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

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

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

The problem of forward declarations (#id54)

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

The problem with type hints is that annotations (per PEP 3107 (/dev/peps/pep-3107), and similar to default values) are evaluated at the time a function is defined, and thus any names used in an annotation must be already defined when the function is being defined. A common scenario is a class definition whose methods need to reference the class itself in their annotations. (More general, it can also occur with mutually recursive classes.) This is natural for container types, for example:

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

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

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

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

A compromise is possible where a `__future__` import could enable turning *all* annotations in a given 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]'}
```

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

(*UPDATE:* That `__future__` import statement and its consequences are discussed in PEP 563 (/dev/peps/pep-0563).)

The double colon (#id55)

A few creative souls have tried to invent solutions for this problem. For example, it was proposed to use a double colon (`::`) for type hints, solving two problems at once: disambiguating between type hints and other annotations, and changing the semantics to preclude 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 is 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 PEP 3107 (/dev/peps/pep-3107) [gvr-artima] (#gvr-artima)). 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 on 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 to 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 code that works with Python 3.5 only. By using existing syntax, the current proposal can easily 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 double colon for argument type hints, for consistency we'd have to use the same convention for future syntax, perpetuating the ugliness.

Other forms of new syntax (#id56)

A few other forms of alternative syntax have been proposed, e.g. the introduction of a `where` keyword [roberge] (#roberge), and Cobra-inspired `requires` clauses. But these all share a problem with the double colon: they won't work for earlier versions of Python 3. The same would apply to a new `__future__` import.

Other backwards compatible conventions (#id57)

The 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 (/dev/peps/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, code 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 notation (`: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 hard to beat (because it was designed for this very purpose).

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

PEP Development Process (#id58)

A live draft for this PEP lives on GitHub [github] (#github). There is also an issue tracker [issues] (#issues), where much of the technical discussion takes place.

The draft on GitHub is updated regularly in small increments. The official PEPS repo [peps (#peps)] is (usually) only updated when a new draft is posted to python-dev.

Acknowledgements (#id59)

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

Influences include existing languages, libraries and frameworks mentioned in PEP 482 (/dev/peps/pep-0482). Many thanks to their creators, in alphabetical order: Stefan Behnel, William Edwards, Greg Ewing, Larry Hastings, Anders Hejlsberg, Alok Menghrajani, Travis E. Oliphant, Joe Pamer, Raoul-Gabriel Urma, and Julien Verlaguet.

References (#id60)

[mypy] (#id1) http://mypy-lang.org (http://mypy-lang.org)

[gvr-artima] *(1 (#id2), 2 (#id6))* http://www.artima.com/weblogs/viewpost.jsp?thread=85551 (http://www.artima.com/weblogs/viewpost.jsp?thread=85551)

[wiki-variance] (#id3) http://en.wikipedia.org/wiki/Covariance_and_contravariance_%28computer_science%29 (http://en.wikipedia.org/wiki/Covariance_and_contravariance_%28computer_science%29)

[typeshed] (#id5) https://github.com/python/typeshed/ (https://github.com/python/typeshed/)

[pyflakes] https://github.com/pyflakes/pyflakes/ (https://github.com/pyflakes/pyflakes/)

[pylint] http://www.pylint.org (http://www.pylint.org)

[roberge] (#id7) http://aroberge.blogspot.com/2015/01/type-hinting-in-python-focus-on.html (http://aroberge.blogspot.com/2015/01/type-hinting-in-python-focus-on.html)

[github] (#id8) https://github.com/python/typing (https://github.com/python/typing)

[issues] (#id9) https://github.com/python/typing/issues (https://github.com/python/typing/issues)

[peps] https://hg.python.org/peps/file/tip/pep-0484.txt (https://hg.python.org/peps/file/tip/pep-0484.txt)

[importdocs] (#id4) https://docs.python.org/3/reference/import.html#submodules (https://docs.python.org/3/reference/import.html#submodules)

Copyright (#id61)

Source: https://github.com/python/peps/blob/master/pep-0484.txt (https://github.com/python/peps/blob/master/pep-0484.txt)

Tweets by @ThePSF

Python Software Foundation @ThePSF

Support the Python community by becoming a #pycon2020 Patron! It comes with a full conference pass and your name on our new "Patron Wall"! Check out all our sponsorship options here: us.pycon.org/2020/sponsors/…

Jan 18, 2020

Python Software Foundation @ThePSF

Start using 2FA and API tokens on PyPI ift.tt/2NwuVUj

Jan 17, 2020

Python Software Foundation @ThePSF

The PSF January Newsletter is out! Sign up now to receive community news: python.org/psf/newsletter/.

PSF Newsletter Signup
The official home of the Python Programming Language
python.org

Embed

View on Twitter

The PSF

The Python Software Foundation is the organization behind Python. Become a member of the PSF and help advance the software and our mission.

Back to Top