

# PEP 484: 类型提示

---

原文 [PEP 484 -- Type Hints](#)

作者

Guido van Rossum (guido@python.org)

Jukka Lehtosalo (jukka.lehtosalo@iki.fi)

Lukasz Langa (lukasz@python.org)

状态 暂定提案

类型 标准类

发布历史

2015年1月16日

2015年3月20日

2015年4月17日

2015年5月20日

2015年5月22日

## 目录

- [摘要](#)
- [理由与目标](#)
- [注解的含义](#)
- [类型定义的语法](#)
- [与函数注释其他用法的兼容性](#)
- [类型注释](#)
- [指定类型](#)
- ['NewType' 工具函数](#)
- [存根文件](#)
- ['typing' 模块](#)
- [Python 2.7 和跨版本代码的建议语法](#)
- [未被接受的替代方案](#)
- [PEP 开发过程](#)
- [参考文献](#)

## 一、摘要

- [PEP 3107](#) 已经引入了函数注解的语法, 但有意将语义保留为未定义. 目前第三方静态类型分析应用工具已经足够多了, 社区人员采用标准用语和标准库中的基线工具就将获益良多.
- 为了提供标准定义和工具, 本 PEP 引入了一个临时模块, 且列出一些不适用于注解情形的约定.

- 需要注意的是, 即使注解符合本规范, 本 PEP 依然明确不会妨碍注解的其他用法, 也不要求(或禁止)对注解的任意特殊处理. 正如 [PEP 333](#) 对 Web 框架的约定, 这只是为了能够更好地合作.
- 例如这个简单函数, 其参数和返回值都在注解中给出了声明:

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

虽然在运行时通过常规的 '`__annotations__`' 属性可以访问到上述注解, 但运行时并不会进行类型检查. 本提案假定存在一个独立的脱机类型检查程序, 用户可以自愿对源代码运行此检查程序. 这种类型检查程序实质上就是一种非常强大的查错工具. (当然某些用户是可以在运行时采用类似的检查程序实现"契约式设计"或JIT优化, 但这些工具尚未完全成熟.)

- 本提案受到 [\[mypy\]](#) 的强烈启发. 如, "整数序列"类型可以写为 '`Sequence[int]`'. 方括号表示无需向语言添加新的语法. 上述示例用到了自定义类型 '`Sequence`', 是从纯 Python 模块 '`typing`' 中导入的. 通过实现元类(metaclass)中的 '`__getitem__()`' 方法, '`Sequence[int]`' 表示法在运行时得以生效(但主要是对脱机类型检查程序有意义).
- 类型系统支持类型组合(Union)、范型类型(Generic type)和特殊类型 '`Any`', '`Any`' 类型可与所有类型相容(即可以赋值给所有类型, 也可以从所有类型赋值). '`Any`' 类型的特性取自渐进定型的理念. 渐进定型和全类型系统已在 [PEP 483](#) 中有所解释.
- 在 [PEP 482](#) 中, 还介绍了其他一些已借鉴或可比较的方案.

## 二、理由与目标

- [PEP 3107](#) 已加入了为函数定义中的各个部分添加注解的支持. 尽管没有为注解定义什么含义, 但已经隐隐有了一个目标, 即把注解用于类型提示 [\[gvr-artima\]](#), 在 [PEP 3107](#) 中这被列为第一个可能应用的场景.
- 本 PEP 旨在为类型注解提供一种标准语法, 让 Python 代码更加开放、更易于静态分析和重构, 提供一种潜在的运行时类型检查方案, 以及(或许在某些上下文中)能利用类型信息生成代码.
- 在这些目标中, 静态分析是最重要的. 包括了对 `mypy` 这类脱机类型检查程序的支持, 以及可供IDE使用的代码自动补全和重构的标准表示法.

### 非本文目标

- 虽然本提案的 '`typing`' 模块将包含一些用于运行时类型检查的功能模块, 特别是 '`get_type_hints()`' 函数, 但必须开发第三程序包才能实现特定的运行时类型检查功能, 比如使用装饰器或元类.
- 还有一点应强调的是, Python 仍将保持为一种动态类型语言, 并且按照惯例作者从未希望让类型提示成为强制类型.

### 三、注解的含义

- 不带注解的函数都应被视为其类型可能是最通用的, 或应被任何类型检查器忽略的. 带有 '@no\_type\_check' 装饰器的函数应被视为不带注解的.
- 建议但并不强求被检查函数的全部参数和返回类型都带有注解. 被检查函数的参数和返回类型的缺省注释为 'Any'. 一个例外是, 实例和类和方法的第一个参数. 如果未带注解, 则假定实例方法的第一个参数类型就是所在类的类型, 而类方法的第一个参数的类型则为所在对象类的类型. 例如在类 A 中, 实例方法的第一个参数的类型隐含为 A, 在类方法中, 第一个参数的精确类型无法用类型注解表示.

(请注意, '\_\_init\_\_' 的返回类型应该用 '-> None' 进行注解. 原因比较微妙. 如果假定 '\_\_init\_\_' 缺省用 '-> None' 做为返回类型注解, 那么是否意味着无参数、不带注解的 '\_\_init\_\_' 方法还需要做类型检查? 与其任其模棱两可或引入异常, 还不如规定 '\_\_init\_\_' 应该带有返回类型注解, 默认表现与其他方法相同.)

- 类型检查程序应对函数主体和所给注解的一致性进行检查. 这些注解还可以用于检查其他被检函数对该函数的调用是否正确.
- 类型检查程序应该尽力推断出尽可能多的信息. 最低要求是能够处理内置装饰器 '@property', '@staticmethod' 和 '@classmethod'.

### 四、类型定义的语法

- 此处的语法充分利用了 PEP 3107 风格的注解以及许多以下章节介绍的拓展. 类型提示的基本格式是把类名填入函数注解的位置:

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

这段代码说明参数 'name' 的预期类型为 'str'. 类似地, 预期的函数返回类型为 'str'.

其类型是特定参数类型子类型的表达式也被该参数接受.

#### 1. 可接受的类型提示

- 类型提示可以是内置类(包含标准库或第三方拓展模块中定义的), 抽象基类, 'types' 模块中提供的类型和用户自定义类(包含标准库或第三方库中定义的).
- 虽然注解通常是类型提示的最佳格式, 但有时更适合用特殊注释或在单独发布的存根文件中表示. (示例见下文.)
- 注解必须是有效的表达式, 其计算过程不会使定义函数时引发异常. (向前引用见下文.)
- 注解应保持简洁, 否则静态分析工具可能无法对其进行解析. 例如动态计算得出的类型可能无法被理解. (此处要求有意模糊, 根据讨论结果可在本 PEP 的未来版本中加入某些包含和排除项.)
- 此外, 以下结构也可用作类型注解: 'None', 'Any', 'Union', 'Tuple', 'Callable', 用于构建由 'typing' 导出的类(如 'Sequence' 和 'Dict')的所有抽象基类 'ABC' 及其替代 'stand-in', 类型变量和类型别名.

- 'typing' 模块提供了所有用于支持以下章节中描述的功能而新引入的类型名(如 'Any' 和 'Union').

## 2. 使用 'None'

- 在类型提示中使用 'None' 时, 'None' 表达式被认为与 'type(None)' 等同.

## 3. 类型别名(Type Aliases)

- 类型别名是通过简单的变量赋值定义的:

```
Url = str

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

需要注意的是, 建议类型别名首字母大写, 因为这代表其是用户自定义类型, 而这种类型(用户自定义类)通常都使用这种方式拼写.

- 类型别名的复杂程度可以和注解中的类型提示一样, 类型注解可接受的内容在类型别名中都可接受.

```
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]
```

这等同于:

```
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[T, T]]
```

## 4. 'Callable'

- 框架需要返回特定签名的回调函数, 则可使用 'Callable[[Arg1Type, Arg2Type], ReturnType]' 形式作为类型提示. 如:

```
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
```

- 在声明返回 'Callable' 类型时也可以不用指定调用签名, 只需用省略号(3个句点)代替参数列表即可:

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

需要注意的是省略号两侧不带括号. 在这种情况下, 回调函数的参数完全没有限制, 且同样可以使用带关键词的参数.

- 由于带关键字参数的回调函数并不常用, 所以当前不支持指定 'Callable' 类型的带关键字参数. 同理, 也不支持参数数量可变的回调函数签名.
- 因为 'type.Callable' 带有双重职能, 用于替代 'collections.abc.Callable', 所以 'isinstance(x, typing.Callable)' 的实现与 'isinstance(x, collections.abc.Callable)' 兼容, 但 'isinstance(x, typing.Callable[...])' 是不受支持的.

## 5. 泛型(Generics)

- 由于容器中的对象类型信息无法以通用方式作出静态推断, 抽象基类已经拓展为支持预约特性, 以表明容器内元素的预期类型. 如:

```
from typing import Mapping, Set

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

- 通过 'typing' 模块中新提供的工厂函数 'TypeVar', 可使得泛型参数化.

```
from typing import Sequence, TypeVar

T = TypeVar("T") # Declare type variable
```

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

本例中约定了返回值的类型与集合内的元素保持一致。

- 'TypeVar()' 表达式只能直接赋值给某个变量(不允许用其组成其他表达式)。
- 'TypeVar()' 的参数必须是一个字符串, 该字符串等于分配给其的变量名。类型变量不允许重新定义。
- 'TypeVar()' 支持把参数可能的类型限为一组固定值(注意: 这里的类型不能用类型变量实现参数化)。
- 'TypeVar()' 支持把采纳数可能的类型限为一组固定值(注意: 这里的类型不能用类型变量实现参数化)。如可以定义某个类型变量只能是 'str' 和 'bytes'。默认情况下, 类型变量会覆盖所有可能的类型。以下是一个约束类型变量范围的示例:

```
from typing import TypeVar, Text

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

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

'concat' 函数对两个 'str' 或两个 'bytes' 参数都可以调用, 但不能混合使用 'str' 和 'bytes' 参数。

只要存在约束条件, 就至少应该有两个, 不允许只指定单个约束条件。

- 在类型变量的的上下文中, 类型变量约束类型的子类型应被视作显式给出的对应基本类型。

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

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

上述调用是合法的, 只是类型变量 'AnyStr' 将被设为 'str' 而非 'MyStr'。实际上, 赋给 'x' 的返回值, 其推断类型也会是 'str'。

- 此外, 'Any' 对于所有类型变量而言都是合法值。

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

上述语句相当于省略了泛型注解, 只写了 'elements: List'。

## 6. 用户自定义的泛型类型

- 把 'Generic' 基类包含进来, 即可将用户自定义类定义为泛型类。如:

```

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))

```

做为基类的 'Generic[T]' 定义了一个带有类型参数 'T' 的 'LoggedVar' 类. 这也使得 'T' 能在类的体内作为类型来使用.

- 'Generic' 基类会用到定义了 '\_\_getitem\_\_' 的元类, 以便 'LoggedVar[t]' 能作为类型来使用:

```

from typing import Iterable

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

```

- 同一个泛型类型所赋的变量类型可以是任意多个, 而且类型变量可以用作约束作用. 以下语句是合法的:

```

from typing import TypeVar, Generic
...

T = TypeVar("T")
S = TypeVar("S")

class Pair(Generic[T, S]):
    ...

```

'Generic' 的每个类型变量参数都必须唯一. 因此以下语句是非法的:

```

from typing import TypeVar, Generic
...

T = TypeVar("T")

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

```

在比较简单的场合, 没有必要用到 'Generic[T]', 这时可以继承其他泛型类并指定类型变量参数:

```

from typing import TypeVar, Iterator

T = TypeVar("T")

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

```

以上类的定义等价于:

```

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

```

- 可以对 'Generic' 使用多重继承:

```

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]):
    ...

```

如未指定类型参数, 则泛型类的子类会假定参数的类型均为 'Any'. 在以下示例中, 'MyIterable' 就不是泛型类, 而时隐式继承自 'Iterable[Any]':



```
from typing import Iterable

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

- 泛型元类不受支持.

## 7. 类型变量的作用域规则

- 类型变量遵循常规的名称解析规则. 但在静态类型检查的上下文中, 存在一些特殊情况:
- 泛型函数中用到的类型变量可以被推断出来, 以便在同一代码块中表示不同的类型. 如:

```
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.
```

- 当泛型类的方法中用到类型变量时, 若该变量正好用作参数化类, 那么此类型变量一定是绑定不变的. 如:

```
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!
```

- 如果某个方法中用到的类型变量与所有用于参数化类的变量都不相符, 则会使得该方法成为返回类型为该类型变量的泛型函数:

```
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.
```

- 在泛型函数体内不应出现未绑定的类型变量, 在类中除方法定义以外的地方也不应出现:

```
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): -> # this is OK though.
        ...
```

- 如果泛型类的定义位于某泛型函数内部, 则其不允许使用参数化该泛型函数的类型变量:

```
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]):
        ...
```

- 嵌套的泛型类不能使用相同的类型变量. 外部类的类型变量, 作用域不会覆盖内部类:

```
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.
```

## 8. 实例化通用类和类型清除

- 当然可以对用户自定义的泛型类进行实例化. 假定编写了以下继承自 'Generic[T]' 的 'Node' 类:

```
from typing import TypeVar, Generic

T = TypeVar("T")

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

- 若要创建 'Node' 的实例, 像普通类一样调用 'Node()' 即可. 在运行时, 实例的类型(类)将会是 'Node'. 但是对于类型检查程序而言, 会要求具备什么类型呢? 答案取决于调用时给出多少信息. 如果构造函数 (\_\_init\_\_ 或 \_\_new\_\_) 在其签名中用了 'T', 且传了相应的参数值, 则会替换对应参数的类型. 否则就假定为 'Any'. 如:

```
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].
```

- 如果推断的类型用了 '[Any]', 但预期的类型更为具体, 则可以用类型注释(参见下文)强行指定变量的类型, 如:

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

或者, 也可以实例化具体的类型, 如:

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

需要注意的是, 'p' 和 'q' 的运行类型(类)仍会保持为 'Node', 'Node[int]' 和 'Node[str]' 是可相互区分的类对象, 但通过实例化创建对象的运行时类不会记录该区别. 这种行为被称为"类型清除". 在 Java, TypeScript 之类的支持泛型的语言中, 这是一种常见做法.

- 通过泛型类(不论是否参数化)访问属性将会导致类型检查失败. 在类定义体之外, 无法对类的属性进行赋值, 它只能通过类的实例访问, 且该实例还不能带有同名的实例属性:

```
# (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.
```

- 类似 'Mapping', 'Sequence' 这种抽象集合类的泛型版本, 以及 'List', 'Dict', 'Set', 'FrozenSet' 这种内置类的泛型版本, 都是不能被实例化的. 但是, 其具体的用户自定义子类 and 具体集合类的泛型版本, 就能被实例化了:

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

注意, 请勿将静态类型和运行时混为一谈. 上述场合中, 类型仍会被清除, 并且以上表达式只是以下语句的简写形式:

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

- 不建议在表达式中直接使用带下标的类(如 'Node[int]'), 最好是采用类型别名(如 'IntNode = Node[int]'). (首先创建 'Node[int]' 这种带下标的类会有一定的运行开销, 其次使用类型别名可读性更好.)

## 9. 用任意泛型类型作为基类

- 'Generic[T]' 只能用作基类, 不适合作为类型来使用. 但上述示例中的用户自定义泛型类型(如 'LinkedList[T]'), 以及内置的泛型类型和抽象基类(如 'List[T]' 和 'Iterable[T]'), 则既可以当作类型使用, 也可当作基类使用. 如, 可以定义带有特定类型参数的 'Dict' 子类:

```
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' 既是 'dict' 的子类, 也是 'Dict[str, List[node]]' 的子类型.

- 如果某个泛型基类带有类型变量作为类型实参, 则会使其定义成为泛型类. 比如可以定义一个既可迭代又是容器的 'LinkedList' 泛型类:

```
from typing import TypeVar, Iterable, Container

T = TypeVar("T")

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

这样 'LinkedList[int]' 就是一种合法的类型. 注意在基类列表中可以多次使用 'T', 只要不再 'Generic[...]' 中多次使用同类型的变量 'T' 即可.

- 再看以下示例:

```
from typing import TypeVar, Mapping
```

```
T = TypeVar("T")

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

以上情况下, 'MyDict' 带有单个参数 'T'.

## 10. 抽象泛型类型

- 'Generic' 使用的元类是 'abc.ABCMeta' 的一个子类. 通过包含抽象方法或属性, 泛型类可以成为抽象基类, 并且泛型类也可以将抽象基类作为基类而不会出现元类冲突.

## 11. 带类型上界的类型变量

- 类型变量可以用 'bound=' 指定类型上界(注意, " 本身不能由类型变量参数化). 这意味着, 替换(显式或隐式)类型变量的实际类型必须是上界类型的子类型. 常见例子就是定义一个 'Comparable' 类型, 这样就注意捕获最常见的错误了:

```
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.
```

需要注意的是, 以上代码还不够理想, 比如 'min("x", 1)' 在运行时是非法的, 但类型检查程序只会推断出返回类型是 'Comparable'. 不幸的是, 解决这个问题需要引入一个强大且复杂得多的概念, F 有界多态性(F-bounded polymorphism). 后续可能还会再来讨论这个问题.

- 类型上界不能与类型约束一起使用(如 'AnyStr' 中的用法, 参见之前的示例), 类型约束会使得推断出的类型一定是约束类型之一, 而类型上界则只要求实际类型是上界类型的子类型.

## 12. 协变和逆变

- 不妨假定有一个 'Employee' 类及其子类 'Manager'. 假如有一个函数, 参数用 'List[Employee]' 做了注解. 那么调用函数时是否该允许使用类型为 'List[Manager]' 的变量作参数呢? 很多人都会不计后果地回答 "是

的, 当然". 但是除非对该函数了解更多信息, 否则类型检查程序应该拒绝此类调用: 该函数可能会在 'List' 中加入 'Employee' 类型的实例, 而这将与调用方的变量类型不符.

- 事实证明, 以上这种参数是有逆变性的, 直观的回答(如果函数不对参数作出修改则没问题!)是要求这种参数具备协变性. 有关这些概念的详细介绍, 参见 Wikipedia [wiki-variance](#) 和 PEP 483. 这里仅演示如何对类型检查程序的行为进行控制.
- 默认情况下, 所有泛型类型的变量均被视作不可变的, 这意味着带有 'List[Employee]' 这种类型注解的变量值必须与类型注解完全相符, 不能是类型参数的子类或超类(上述示例中即为 'Employee').
- 为了便于声明可接受协变或逆变类型检查的容器类型, 类型变量可带有关键字参数 'covariant=True' 或 'convariant=True'. 两者只能有一个. 如果泛型类型带有此类变量定义, 则其变量会被相应视为具备协变或逆变性. 按照约定, 建议对带有 'covariant=True' 定义的类型变量命名时采用 '\_co' 结尾, 而对于带有 'convariant=True' 定义的类型变量则以 '\_contra' 结尾来命名.
- 以下典型示例将会定义一个不可修改或只读的容器类:

```
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
```

- 'typing' 中的只读集合类都将类型变量声明为可协变的, 比如 'Mapping' 和 'Sequence'. 可修改的集合类(如 'MutableMapping' 和 'MutableSequence')则声明为不可变的. 协变类型的一个例子是 'Generator' 类型, 其 'send()' 的参数类型是可协变的(参加下文).
- 需要注意的是协变和逆变并不是类型变量的特性, 而是用该变量定义的泛型类的特性. 可变性仅适用于泛型类型, 泛型函数则没有此特征. 泛型函数只允许采用不带 'covariant' 和 'convariant' 关键字参数的类型变量进行定义. 如:

```
from typing import TypeVar
```

```

class Employee: ...

class Manager(Employee): ...

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

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

dump_employee(Manager()) # OK

```

而以下写法是不可以的:

```

B_co = TypeVar("B_co", covariant=True)

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

```

### 13. 数值类型的继承关系

- [PEP 3141](#) 定义了 Python 的数值类型沉寂关系, 并且 `stdlib` 的模块 `'numbers'` 实现了对应的抽象基类(`'Number'`, `'Complex'`, `'Real'`, `'Rational'` 和 `'Integral'`). 关于这些抽象基类是存在一些争议, 但内置的具体实现的数值类 `'complex'`, `'float'` 和 `'int'` 已得以广泛应用(尤其是后两个类).
- 本 PEP 提出了一种简单、快捷、几乎也是高效的方案, 用户不必先写 `'import numbers'` 语句再使用 `'numbers.float'`: 只要注解为 `'float'` 类型, 即可接受 `'int'` 类型的参数. 类似地, 注解为 `'complex'` 类型的参数, 则可接受 `'float'` 或 `'int'` 类型. 这种方案无法应对实现抽象基类或 `'Fractions.Fraction'` 类的类, 但可以相信那些用户场景极为罕见.

### 14. 向前引用

- 当类型提示包含尚未定义的名称时, 未定义名称可以先表示为字符串字面量, 稍后再作解释.
- 在定义容器类时, 通常就会发生这种情况, 这时在某些方法的签名中会出现将要定义的类. 如, 以下代码(简单的二叉树实现的开始部分)将无法生效:

```

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

```

为了解决问题, 可以写为:



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

此字符串字面量应包含一个合法的 Python 表达式, 即 `'compile(lit, "", "eval")'` 应该是有效的代码对象, 并且在模块全部加载完成后对其求值应该不会出错. 对该表达式求值时所处的局部和全局命名空间应对同一函数的默认参数求值时的命名空间相同.

此外, 该表达式应可被解析为合法的类型提示, 即受限于 "可接受的类型提示" 一节中的规则约束.

- 允许将字符串字面量作为类型提示的一部分, 如:

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

- 向前引用的常见应用场景是签名需要用到 Django 模型. 通常, 每个模型都存放在单独的文件中, 并且模型有一些方法的参数类型会涉及到其他的模型. 因为 Python 存在循环导入处理机制, 往往不能直接导入所有要用到的模型:

```
# 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
```

假定先导入了 'main', 则 'models/b.py' 的 'from models.a import A' 一行将会运行失败, 报错 'ImportError', 因为在 'a' 定义类 'A' 之前就打算从 'models/a.py' 导入它. 解决办法是换成只导入模块, 并通过 '`module.class`' 名引用 'models':

```
# 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
```

## 15. 'Union' 类型

- 因为一个参数可接受数量有限的几种预期类型是常见需求, 所以系统新提供了一个特殊的工厂类, 名为 'Union'. 如:

```
from typing import Union

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

- 'Union[T1, T2, ...]' 生成的类型是所有 'T', 'T2' 等类型的超级类型, 因此只要是这些类型之一的值就可被 'Union[T1, T2, ...]' 注解的参数所接受.
- 'Union' 类型的一种常见情况是 'Optional' 类型. 除非函数定义中提供了默认值 'None', 否则 'None' 默认是不能当任意类型的值使用. 如:

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

'Union[T1, None]' 可以简写为 'Optional[T1]', 比如以上语句等同于:

```
from typing import Optional

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

- 本 PEP 以前允许类型检查程序在默认值为 'None' 时假定采用 'Optional' 类型, 如下所示:

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

将被视为等效于:

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

现在不再推荐这种做法. 类型检查程序应与时俱进, 将需要 'Optional' 类型的地方明确指出来.

## 16. 用 'Union' 实现单实例类型的支持

- 单实例通常用于标记某些特殊条件, 特别是 'None' 也是合法变量值的情况下. 如:

```
_empty = object()

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

- 为了在这种情况下允许精确设定类型, 用户应结合使用 'Union' 类型和标准库提供的 'enum.Enum' 类, 这样就能静态捕获类型错误了:

```
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
        return x * 2 # that can only have type int.
```

- 因为 'Enum' 的子类无法被继承, 所以在上述示例的所有分支中都能静态推断出变量 'x' 的类型. 需要多种单例对象的情形也同样适用, 可以适用包含多个值的枚举:

```
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
```

## 17. Any 类型

- 'Any' 是一种特殊的类型. 每种类型都与 'Any' 相符. 可以将其视为包含所有值和所有方法的类型. 需要注意的是, 'Any' 和内置类型对象完全不同.
- 当某个值的类型为 'object' 时, 类型检查程序将拒绝几乎所有对其进行的操作, 将其赋给类型更具体的变量(或将其用作返回值)将是一种类型错误. 反之, 当值的类型为 'Any' 时, 类型检查程序将允许对其执行的所有操作, 并且 'Any' 类型的值可以赋给类型更具体的变量(或用作返回值).
- 不带类型注解的函数参数假定就是用 'Any' 作为注解的. 如果用了泛型类型但又未指定类型参数, 则也假定参数类型为 'Any':

```
from typing import Mapping

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

上述规则也适用于 'Tuple', 在类型注解的上下文中, 'Tuple' 等效于 'Tuple[Any, ...]', 即等效于 'tuple'. 同样, 类型注解中的 'Callable' 等效于 'Callable[..., Any]', 即等效于 '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:
    ...
```

## 18. 'NoReturn' 类型

- 'typing' 模块提供了一种特殊的类型 'NoReturn', 用于注解一定不会正常返回的函数. 如一个将无条件引发异常的函数:

```
from typing import NoReturn

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

- 类型注解 'NoReturn' 用于 'sys.exit' 之类的函数. 静态类型检查程序将会确保返回类型注解为 'NoReturn' 的函数确实不会隐式或显式地返回:

```
import sys
from typing import NoReturn

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

- 类型检查程序还会识别出调用此类函数后面的代码是否可达, 并采取相应动作:

```
# Continue from first example.
def g(x: int) -> int:
    if x > 0:
        return x
    stop()
    return "whatever works" # Error might be nor reported by some
                            # that ignore errors in unreachable
blocks.
```

- 'NoReturn' 类型仅可以用于函数的返回类型注解, 出现在其他位置则被认为是错误:

```
from typing import List, NoReturn
```

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

## 19. 类对象的类型

- 有时会涉及到类对象, 特别是从某个类继承而来的类对象. 类对象可被写为 'Type[C]', 这里的 'C' 是一个类. 为了清楚起见, 'C' 在用作类型注解时指的是类 'C' 的实例, 'Type[C]' 指的是 'C' 的子类. 这类似于对象和类型之间的区别. 如, 假设有以下类:

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

- 假设有一个函数, 如果传一个类对象进去, 就会创建出该类的一个实例:

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

- 若不用 'Type[]', 能给 'new\_user()' 加上的最好的类型注解将会是:

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

- 但采用 'Type[]' 和带上界的类型变量, 就可以注解得更好:

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

- 现在, 若在 'user' 的某个子类做参数调用 'new\_user()', 类型检查程序将能推断出结果的正确类型:

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

- 'Type[C]' 对应的值必须是类型为 'C' 的子类型的类对象实体, 而不是某个具体的类型. 换句话说, 在上述示例中, 'new\_user(Union[BasicUser, ProUser])' 之类的调用将被类型检查程序拒绝(并且会运行失败, 因为 'Union' 无法实例化).
- 请注意, 用类的 'Union' 作 'Type[]' 的参数是合法的, 如下所示:

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

但是在运行时上例传入的实际参仍必须是具体的类对象:

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

- 'Type[Any]' 也是支持的, 含义常见下文.
- 为类方法的第一个参数标注类型注解时, 允许采用 'Type[T]', 这里的 'T' 是一个类型变量, 具体请参阅相关章节.
- 任何其他结构(如 'Tuple' 或 'Callable')均不能用作 'Type' 的参数.
- 此特性存在一些问题: 比如若 'new\_user()' 要调用 'user\_class()', 就意味着 'User' 的所有子类都必须在其构造函数的签名中支持该调用. 不过并不是只有 'Type[]' 才会如此, 类方法也有类似的问题. 类型检查程序应该将违反这种假定的行为标记出来, 但与所表明基类(如上例中的 'User')的构造函数签名相符的构造函数, 应该默认是允许调用的. 如果程序中包含了比较复杂的或可拓展的类体系, 也可以采用工厂类方法来作处理. 本 PEP 的未来修订版本可能会引入更好的方法来解决这些问题.
- 当 'Type' 带有参数时, 仅要求有一个参数. 不带中括号的普通类型等效于 'Type[Any]', 也即等效于 'type' (Python 元类体系中的根类). 这种等效性也促成了其名称 'Type', 而没有采用 'Class' 或 'SubType' 这种名称, 在讨论此特性时这些名称都被提出过, 这有些类似 'List' 和 'list' 的关系.
- 关于 'Type[Any]' (或 'Type', 'type') 的行为, 如果要访问该类型变量的属性, 则只提供了 'type' 定义的属性和方法(如 '\_\_repr\_\_()' 和 '\_\_mro\_\_'). 此类变量可以用任意参数进行调用, 返回类型则为 'Any'.
- 'Type' 的参数是协变的, 因为 'Type[Derived]' 是 'Type[Base]' 的子类型:

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

## 20. 为实例和类方法加类型注释

- 大多数情况下, 类和实例方法的第一个参数不需要加类型注解, 对实例方法而言假定它的类型就是所在类(的类型), 对类方法而言它则是所在类对象对应的类型对象(的类型). 另外, 实例方法的第一个参数加类型

注解时可以带有一个类型变量. 这时返回类型可以采用相同的类型变量, 从而使该方法成为泛型函数. 如:

```
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.
```

- 同时, 可以对类方法第一个参数的类型注解中使用 'Type[]':

```
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.
```

请注意, 某些类型检查程序可能对以上用法加以限制, 比如要求所用类型变量具备合适的类型上界(参见实例).

## 21. 版本和平台检查

- 类型检查程序应该能理解简单的版本和平台检查语句, 如:

```
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.
```

- 不要指望类型检查程序可以理解诸如 `"".join(reversed(sys.platform)) == "xunil"` 这种晦涩的语句.



## 22. 运行时检查还是类型检查

- 有时候, 有些代码必须由类型检查程序(或其他静态分析工具)进行检查, 而不应拿去运行. 'typing' 模块为这种情况定义了一个常量 'TYPE\_CHECKING', 在类型检查(或其他静态分析)期间视其为 'True', 在运行时视其为 'False'. 如:

```
import typing

if typing.TYPE_CHECKING:
    import expensive_mod

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

注意这里的类型注解必须用引号引起来, 使其成为 "向前引用", 以便向解释器隐藏 'expensive\_mod' 引用. 在 '# type' 注释中无需加引号.

这种做法对于处理循环导入也会比较有用.

## 23. 可变参数列表和默认参数值

- 可变参数列表也可以加类型注解, 以下定义是可行的:

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

这表示以下函数调用的参数类型都是合法的:

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

在 'foo' 函数体中, 变量 'args' 的类型被推导为 'Tuple[str, ...]', 变量 'kwds' 的类型被推导为 'Dict[str, int]'.

- 在存根文件中, 将参数声明为带有默认值, 但不指定实际的默认值, 这会很有用. 如:

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

默认值应该是如何的? ""、'b' 或 'None' 都不符合类型约束.

这时可将默认值指定为省略号, 即为以上示例.

## 24. 只采用位置参数

- 有些函数被设计成只能按位置接受参数, 并希望调用者不要使用参数名称, 不通过关键字给出参数. 名称以 '\_' 开头的参数均被假定为只按位置访问, 除非同时以 '\_' 结尾:

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

quux(3, __y__=1) # This call is fine.
quux(__x=3)      # This call is an ERROR.
```

## 25. 为生成器函数和协程加类型注解

- 生成器函数的返回类型可以用 'typing' 模块提供的泛型 'Generator[yield\_type, send\_type, return\_type]' 进行类型注解:

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

- PEP 492 中引入的协程(coroutine)可用与普通函数相同的语法进行类型注解. 但是返回类型的类型注解对应的是 'await' 表达式的类型, 而非协程的类型:

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

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

- 'typing' 模块提供了一个抽象基类 'collections.abc.Coroutine' 的泛型版本, 以支持可异步调用特性, 同时支持 'send()' 和 'throw()' 方法. 类型变量定义及其顺序与 'Generator' 的相对应, 即 'Coroutine[T\_co, T\_contra, V\_co]', 如:

```
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
```

该模块还为无法指定更精确类型的情况提供了泛型抽象基类 'Awaitable'、'AsyncIterable' 和 'AsyncIterator':

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

## 五、与函数注释其他用法的兼容性

- 有一些函数注解的使用场景, 与类型提示是不兼容的. 这些用法可能会引起静态类型检查程序的混乱. 但因为类型提示的注解在运行时不起作用(计算注解表达式、将注解存储在函数对象的 '.\_\_annotations\_\_' 属性中除外), 所以不会让程序报错, 只是可能会让类型检查程序发出虚报警告或错误.
- 如果要想某部分程序不受类型提示的影响, 可以哟哦嗯以下一种或几种方法进行标记:
  - 用 '# type: ignore' 加注解;
  - 为类或函数加上 '@no\_type\_check' 装饰符;
  - 为自定义类或函数装饰符加上 '@no\_type\_check\_decorator' 标记.

更多详情, 请参加后续章节.

- 为了最大程度与脱机类型检查过程保持兼容, 将依赖于类型注解的接口改成其他机制(如装饰器)可能比较合适些. 不过这在 Python 3.5 中没有什么关系. 更多讨论请参见后续的 "未被采纳的其他方案".

## 六、类型注释

- 本 PEP 并未将变量明确标为某类型提供一等语法支持. 为了有助于在复杂情况下类型推断, 可以采用以下格式的注释:

```
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]
```

- 类型注释应放在变量定义语句的最后一行, 还可以紧挨着冒号放在 'with' 和 'for' 语句后面.

以下是 'with' 和 'for' 语句的类型注解示例:

```
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.
    ...
```

- 在存根文件中, 只声明变量的存在但不给出初值可能会比较有用. 这用 [PEP 526](#) 的变量注解语法即可实现:

```
from typing import IO

stream: IO[str]
```

上述语法在所有版本的 Python 的存根文件均可接受. 但在 Python 3.5 以前版本的非存根文件代码中, 存在一种特殊情况:

```
from typing import IO

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

- 尽管 'None' 与给定类型不符, 类型检查程序不对应上述语句报错, 也不应将类型推断结果更改为 'Optional[...]'(虽然规则要求对注解默认值为 'None' 的参数如此操作). 这里假定将由其他代码保证赋予变量类型合适的值, 并且所有调用都可假定该变量具有给定类型.

注释 '# type: ignore' 应该放在错误信息所在行上:

```
import http.client

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

如果注释 '# type: ignore' 位于文件的开头、单独占一行、在所在文档字符串、'import' 语句或其他可执行之前, 则会让文件中所有错误都不报错. 空行和其他注释(如 shebang 代码行和编码 cookie)可以出现在 '# type: ignore' 之前.

- 某些时候, 类型注释可能需要与查错工具或其他注释同处一行. 此时类型注释应位于其他注释和 lint 标记之前:

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

- 如果大多数时候类型提示能被证明有用, 那么将来版本的 Python 可能会为 'typing' 变量提供语法. (更新: 该语法以通过 [PEP 526](#) 在 Python 3.6 加入.)

## 七、指定类型

- 偶尔, 类型检查程序可能需要另一种类型提示: 程序员可能知道, 某个表达式的类型检查程序能够推断出来的更为准确. 如:

```
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])
```

- 某些类型检查程序可能无法推断出 'a[index]' 的类型为 'str', 而只能推断出是个对象或 'Any', 但大家都知道(如果代码能够运行到该点)它必须是个字符串. 'ast(t, x)' 调用会通知类型检查程序, 确信 'x' 的类型就是 't'. 在运行时, 'cast' 始终会原封不动地返回表达式, 不做类型检查, 也不对值作任何转换或强制转换.
- 'cast' 与类型注释不同. 用了类型注释, 类型检查程序仍应验证推断出的类型是否与声明的类型一致. 若用了 'cast', 类型检查程序就会完全信任程序员. 'cast' 还可以在表达式中使用, 而类型注释则只能在赋值是使用.

## 八、'NewType' 工具函数

- 还有些时候, 为了避免逻辑错误, 程序员可能会创建简单的类. 如:

```
class UserId(int):
    pass

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

- 但创建类会引入运行时的开销. 为了避免这种情况, 'typing.py' 提供了一个工具函数 'NewType', 该函数能够创建运行开销几乎为零的唯一简单类型. 对于静态类型检查程序而言, 'Derived = NewType("Derived", Base)' 大致等同于以下定义:

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

在运行时, 'NewType("Derived", Base)' 将返回一个伪函数, 该伪函数只是简单地将参数返回. 类型检查程序在用到 'UserId' 时要求由 'int' 显式转换而来, 而用到 'int' 时要求由 'UserId' 显式转换而来. 如:

```
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' 只能接受两个参数: 新的唯一类型名称、基类. 后者应为合法的类(即不是 'Union' 这种类型结构), 或者是通过调用 'NewType' 创建的其他唯一类型. 'NewType' 返回的函数仅接受一个参数, 这等同于仅支持一个构造函数, 构造函数只能接受一个基类实例作参数(参见上文). 如:

```
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.
```

- 对 'NewType("Derived", Base)' 进行 'isinstance'、'issubclass' 和派生子类的操作都会失败, 因为函数对象不支持这些操作.

## 九、存根文件

## 十、'typing' 模块

- 为了将静态类型检查特性开放给 Python 3.5 以下的版本使用, 需要有一个统一的命名空间. 为此, 标准库中引入了一个名为 'typing' 的新模块.
- 'typing' 模块定义了用于构建类型的构件(如 'Any')、表示内置集合类的泛型变体类型(如 'List')、表示集合类的泛型抽象基类类型(如 'Sequence')和一批便捷类定义.

需要注意的是, 只有在类型注解的上下文中才支持用 'TypeVar' 定义的特殊类型结构, 比如 'Any'、'Union' 和类型变量, 而 'Generic' 则只能用作基类. 如果出现在 'isinstance' 或 'issubclass' 中, 所有这些(未参数化的泛型除外)都将引发 'TypeError' 异常.

- 基础构件:
  - 'Any', 用法为 'def get(key: str) -> Any: ...'.
  - 'Union', 用法为 'Union[Type1, Type2, Type3]'.
  - 'Callable', 用法为 'Callable[[Arg1Type, Arg2Type], ReturnType]'.
  - 'Tuple', 用于列出元素类型, 比如 'Tuple[int, int, str]'. 空元组类型可以表示为 'Tuple[()]'. 可变长同构元组可以表示为一个类型和省略号, 比如 'Tuple[int, ...]', 此处的 '...' 是语法的组成部分.
  - 'TypeVar', 用法为 'x = TypeVar("x", Type1, Type2, Type3)' 或简化为 'Y = TypeVar("Y")'(详见上文).
  - 'Generic', 用于创建用户自定义泛型类.
  - 'Type', 用于对类对象做类型注解.
- 内置集合类的泛型变体:
  - 'Dict', 用法为 'Dict[key\_type, value\_type]'.
  - 'DefaultDict', 用法为 'DefaultDict[key\_type, value\_type]', 是 'collections.defaultdict' 的泛型变体.
  - 'List', 用法为 'List[element\_type]'.
  - 'Set', 用法为 'Set[element\_type]'. 参阅下文有关 'AbstractSet' 的备注信息.
  - 'FrozenSet', 用法为 'FrozenSet[element\_type]'.

注意, 'Dict'、'DefaultDict'、'List'、'Set' 和 'FrozenSet' 主要用于对返回值做类型注解. 而函数参数的注解, 建议采用下述抽象集合类型, 比如 'Mapping'、'Sequence' 或 'AbstractSet'.

- 容器类抽象基类的泛型变体(及一些非容器类):
  - 'Awaitable'.
  - 'AsyncIterable'.
  - 'AsyncIterator'.
  - 'ByteString'.
  - 'Callable'(详见上文).
  - 'Collection'.
  - 'Container'.
  - 'ContextManager'.
  - 'Coroutine'.
  - 'Generator', 用法为 'Generator[yield\_type, send\_type, return\_type]', 表示生成器函数的返回值. 此为 'Iterable' 的子类型. 并且为 'send()' 方法可接受的类型加入了类型变量(可逆变). 可逆变的意思是, 在要求可发送 'Manager' 实例的上下文中, 生成器允许发送 'Employee' 实例, 并返回生成器的类型.
  - 'Hashable'(非泛型).
  - 'ItemsView'.
  - 'Iterable'.
  - 'Iterator'.
  - 'KeysView'.
  - 'Mapping'.

- 'MapView'.
- 'MutableMapping'.
- 'MutableSequence'.
- 'MutableSet'.
- 'Sequence'.
- 'Set', 重命名为 'AbstractSet'. 因为 'typing' 模块中的 'Set' 表示泛型 'set()', 所以需要改名.
- 'Sized'(非泛型).
- 'ValuesView'.
- 一些用于测试某个方法的一次性类型, 类似于 'Hashable' 或 'Sized':
  - 'Reversible', 用于测试 '\_\_reversed\_\_'.
  - 'SupportsAbs', 用于测试 '\_\_abs\_\_'.
  - 'SupportsComplex', 用于测试 '\_\_complex\_\_'.
  - 'SupportsFloat', 用于测试 '\_\_float\_\_'.
  - 'SupportsInt', 用于测试 '\_\_int\_\_'.
  - 'SupportsRound', 用于测试 '\_\_round\_\_'.
  - 'SupportsBytes', 用于测试 '\_\_bytes\_\_'.
- 便捷类定义:
  - 'Optional', 定义为 'Optional[t] == Union[t, None]'.
  - 'Text', 只是 Python 3 中 'str'、Python 2 中 'unicode' 的别名.
  - 'AnyStr', 定义为 'TypeVar("AnyStr", Text, bytes)'.
  - 'NamedTuple', 用法为 'NamedTuple(type\_name, [(field\_name, field\_type), ...])' 等价于 'collections.namedtuple(type\_name, [field\_name, ...])'. 在为命名元组类型的字段进行类型声明时, 这会很有用.
  - 'NewType', 用于创建运行开销很小的唯一类型, 如 'UserId = NewType("UserId", int)'.
  - 'cast()', 如前所述.
  - '@no\_type\_check', 用于禁止对某个类或函数做类型检查的装饰器(参加下文).
  - '@no\_type\_check\_decorator', 用于创建自定义装饰器的装饰器, 含义与 '@no\_type\_check' 相同(参见下文).
  - '@type\_check\_only', 仅在对存根文件做类型检查时可用的装饰器, 标记某个类或函数在运行时不可用.
  - '@overload', 如上所述.
  - 'get\_type\_hints()', 用于获取函数或方法的类型提示信息的工具函数. 给定一个函数或方法对象, 它将以 '\_\_annotations\_\_' 的格式返回一个 'dict', 向前引用将在原函数或方法定义的上下文进行表达式求值.
  - 'TYPE\_CHECKING', 运行时为 'False', 而对类型检查器则为 'True'.
- I/O 相关的类型:
  - 'IO'(基于 'AnyStr' 的泛型).
  - 'BinaryIO'(只是 'IO[bytes]' 子类型).
  - 'TextIO'(只是 'IO[str]' 子类型).
- 与正则表达式和 're' 模块相关的类型:
  - 'Match' 和 'Pattern', 're.match()' 和 're.compile()' 的结果类型(基于 'AnyStr' 的泛型).



## 十一、Python 2.7 和跨版本代码的建议语法

- 某些工具软件可能在必须与 Python 2.7 兼容的代码中支持类型注解. 为此, 本 PEP 在此给出建议性(而非强制)扩展, 其中函数的类型注解放入 '# type' 注释中. 这种注释必须紧挨着函数头之后, 但在文档字符串之前. 举个例子, 下述 Python 3 代码:

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

等价于以下代码:

```
def embezzle(self, account, funds=100000, *fake_receipts):
    # type: (str, int, *str) -> None
    """Embezzle funds from account using fake receipts."""
    <code goes here>
```

请注意, 方法的 'self' 不需要注明类型.

- 无参数方法则应如下所示:

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

- 有时需要仅为函数或方法指定返回类型, 而暂不指定参数类型. 为了明确这种需求, 可以用省略号替换参数列表. 如:

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

- 参数列表有时会比较长, 难以用一条 '# type:' 注释来指定类型, 为此可以每行给出一个参数, 并在每个参数的逗号之后加上必要的 '# type:' 注释. 返回类型可以用省略号语法指定. 指定返回类型不是强制性要求, 也不是每个参数都需要指定类型. 带有 '# type:' 注释的行应该只包含一个参数. 最后一个参数的类型注释应该在右括号之前. 如:

```
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>
```

- 注意事项:

- 只要工具软件支持这种类型注释语法, 就应该与 Python 版本无关. 为了支持横跨 Python 2 和 Python 3 的代码, 必须如此.
- 参数或返回值不得同时带有类型注解(annotation)和类型注释(comment).
- 如果要采用简写格式(如 '#type: (str, int) -> None'), 则每一个参数都必须如此, 实例和类方法的第一个参数除外. 这第一个参数通常会省略注释, 但也允许带上.
- 简写格式必须带有返回类型. 如果是 Python 3 则会省略某些参数或返回类型, 而 Python 2 则应使用 'Any'.
- 采用简写格式时, '\*args' 和 '\*\*kwargs' 的类型注解前面请对应放置一或二个星号(\*). 在用 Python 3 注解格式时, 此处的注解表示的是每个参数值的类型, 而不是由特殊参数值 'args' 或 'kwargs' 接受到的 'tuple' / 'dict' 的类型.
- 与其他的类型注释相类似, 类型注解中用到的任何名称都必须由包含注解的模块导入或定义.
- 采用简写格式时, 整个注解必须在一行之内.
- 简写格式也可以与右括号处于同一行, 如:

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

- 类型检查程序会将位置不对的类型注释标记为错误. 如有必要, 可以对此类注释作两次注释标记. 如:

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

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

- 在对 Python 2.7 代码做类型检查时, 类型检查程序应将 'int' 和 'long' 视为相同类型. 对于注解为 'Text' 的参数, 'str' 和 'unicode' 类型也应该是可接受的.

## 十二、未被接受的替代方案

- 在讨论本 PEP 的早期草案时, 出现过各种反对意见, 并提出过一些替代方案. 在此讨论其中一些意见, 并解释一下未被接受的原因.
- 下面是几个主要的反对意见.

### 1. 泛型参数该用什么括号?

- 大多数人都知道在 C++、Java、C# 和 Swift 等语言中, 用尖括号(如 'List')来表示泛型的参数化. 这种格式的问题是难以解析, 尤其是对于像 Python 这种思维简单的解析器而言. 在大多数语言中, 通常只允许在特定的语法位置用尖括号来解决歧义, 而这些位置不允许出现泛型表达式. 并且还得采用非常强大的解析技术, 可对任何一段代码进行重复解析.
- 但在 Python 中, 更愿让类型表达式(在语法上)与其他表达式一样, 以便使用变量赋值之类的操作就能创建类型别名. 不妨看看下面这个简单的类型表达式:

```
List<int>
```

从 Python 解析程序的角度来看, 以 4 个短语(名称、小于、名称、大于)开头的表达式将视为连续比较:

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

甚至可以创建一个两种解析方式共存的示例:

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

假设语言中包含尖括号, 则以下两种解释都是可以的:

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

当然能够再提出一种规则来消除上述情况的歧义, 但对于大多数用户而言, 会觉得这些规则稍显随意和复杂. 并且这还要将 CPython 解析程序(和其他所有 Python 解析程序)做出很大的改动. 有一点应该注意, Python 当前的解析程序是有意如此 "愚蠢" 的, 这样用户很容易就想到简单的语法.

因为上述所有原因, 所以方括号(如 'List[int]')是(长期以来都是)泛型参数的首选语法. 这通过在元类上定义 '`__getitem__()`' 方法就可以实现, 根本不需要引入新的语法. 这种方案在所有较新版本的 Python(从 Python 2.2 开始)中均有效. 并非只有 Python 才选择这种语法, Scala 中的泛型类也采用了方括号.

## 2. 已在用的注解怎么办

- 有一条观点指出, [PEP 3107](#) 明确支持在函数注解中使用任意表达式. 因此, 本条新提案被认为与 [PEP 3107](#) 规范不兼容.
- 对此的回应是, 首先本提案没有引入任何直接的不兼容性, 因此使用注解的程序在 Python 3.4 中仍然可以正确运行, 在 Python 3.5 中也毫无影响.
- 类型提示确实期望能最终成为注解的唯一用途, 但这需要再多些讨论, 而且 Python 3.5 才首次推出 'typing' 模块, 也需要一段时时间实现废弃. 直至 Python 3.6 发布之前, 当前的 PEP 都将为临时状态(参阅 [PEP 411](#)). 可能的方案最快将自 3.6 开始无提示地废弃非类型提示的注解, 自 3.7 开始完全废弃, 并在 Python 3.8 中将类型提示声明为唯一允许使用的注解. 即便类型提示在一夜之间取得了成功, 也应该让带注解程序包的作者有足够时间去更换方案.

(更新: 2017年秋季, 本 PEP 和 'type.py' 模块的临时状态终止计划已作更改, 因此其他注解用法的弃用计划也已更改. 更新过的时间计划请参阅 [PEP 563](#))

- 另一种可能的结果是, 类型提示最终将成为注解的默认含义, 但将其禁用的选项也会一直保留. 为此, 本提案定义了一个装饰器 '@np\_type\_check', 该装饰器将禁止对给定类或函数中用作类型提示的注解作默认解释. 这里还定义了一个元装饰器 '@no\_type\_check\_decorator', 可用于对装饰器进行装饰, 使得用其装饰的任何函数或类中的注解都会被类型检查程序忽略.

而且还有 '# type: ignore' 注解, 静态检查程序应支持对选中包禁止类型检查的配置项.

- 尽管有这么多选择可用, 但允许类型提示和其他形式的注解共存于参数中的提案已经发布过了一些. 有一项提案建议, 如果某个参数的注解是字典字面量, 则每个字典键都表示一种不同格式的注解, 字典键 'type' 将用于类型提示. 这种想法及其变体的问题在于, 注解会变得非常 "杂乱", 可读性很差. 而且大多数现有采用注解的库, 几乎不需要与类型提示混合使用. 因此, 只要有选择地禁用类型提示就足够了.

## 3. 前向声明的问题

- 当类型提示必须包含向前引用时, 当前提案无疑是次优选择. Python 要求所有变量在 "用到" 时再做定义. 除了循环导入外, 这不会存在问题: 这里的 "用到" 表示 "运行时查找", 并且对大多数 "向前" 引用而言, 确保在用到名称的函数被调用之前定义名称就没有问题.
- 类型提示的问题在于, 在定义函数时会对注解进行求值(根据 [PEP 3107](#), 类似于默认值), 因此注解中使用的任何名称在定义函数时必须已经定义. 常见的场景是类的定义, 其方法需要在注解中引用类本身. 更一般地说, 在相互递归引用的类中也可能发生这种情况. 对于容器类型而言这很自然, 如:

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

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

上述写法是行不通的, 因为 Python 的特性就是, 一旦类的全体代码执行完毕, 类的名称就定义完成了. 我们的解决方案不是特别优雅, 但是可以完成任务, 也就是允许在注解中使用字符串字面量. 不过大多数时候都

不必用到字符串, 类型提示的大多数应用都应引用内置类型或其他模块中已经定义的类型.

- 有一种答复将会修改类型提示的语义, 以便根本不会在运行时对其进行求值. 毕竟类型检查是脱机进行的, 为什么要在运行时对类型提示进行求值呢. 当然这与向下兼容有冲突, 因为 Python 解释程序其实并不知道某个注解时是要用作类型提示或是有其他用途.
- 有一种可行的折衷方案, 就是用 '`__future__`' 导入可以将给定模块中的所有注解都转换为字符串字面量, 如下所示:

```
from __future__ import annotations


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

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

这种 '`future`' 导入语句可能会在单独的 PEP 中给出.

(更新: [PEP 563](#) 中已经讨论了 '`__future__`' 导入语句及其后果.)

#### 4. 双冒号

- 一些有创造力的人已经尝试发明了多种解决方案. 比如有些人提议让类型提示采用双冒号(, 可以一次解决两个问题: 消除类型提示与其他注解之间的歧义、修改语义避免运行时求值. 但这种想法有以下几个问题.
  - 丑陋. 在 Python 中单个冒号有很多用途, 并且看起来很熟悉, 因为类似于英文中的冒号用法. 这是一条普遍的经验法则, Python 会遵守标点符号的大多数使用格式, 那些例外通常也是因其他编程语言而熟知的. 但是 `::` 的这种用法在英语中是闻所未闻的, 而在其他语言(如 C++)中是被用作作用域操作符的, 这太与众不同了. 反而, 类型提示采用单个冒号读起来很自然, 这不足为奇, 因为这是为此目的而精心设计的(想法比 [PEP 3107 \[gvr-artima\]](#) 要早的多). 从 Pascal 到 Swift, 其他很多语言也采用了相同的风格.
  - 那该如何处理返回类型的注解呢?
  - 实际上这是在运行时对类型提示进行求值的特性.
    - 让类型提示可用于运行时, 使得能基于类型提示构建运行时的类型检查程序.
    - 即便代码尚未运行, 类型检查程序仍能捕获错误. 因为类型检查程序是一个单独的程序, 所以用户可以选择不运行(甚至不安装), 但仍可能想把类型提示用作简明的文档. 错误的类型提示即便当作文档也没啥用处.
  - 因为是新语法, 所以把双冒号用于类型提示将会受到限制, 只能适用于 Python 3.5 的代码. 而利用现有语法, 本提案可以轻松应用于较低版本的 Python 3. mypy 实际支持 Python 3.2 以上的版本.
  - 如果类型提示获得成功, 可能会决定在未来加入的新语法, 用于声明变量的类型, 比如 '`var age: int = 42`'. 如果参数的类型提示采用双冒号, 那么为了保持一致, 未来的语法必须采用同样的约定, 如此难看的语法将会一致流传下去.

## 5. 其他一些新语法格式

- 还有一些其他格式的语法也被提出过, 比如 'where' 关键字的引入, 以及 Cobra-inspired 'requires' 子句. 但这些语法都和双冒号一样存在同样的问题, 他们不适用于低版本的 Python 3. 新的 '\_\_future\_\_' 导入语法也同样如此.

## 6. 其他的向下兼容约定

- 提出的想法有:
  - 装饰器, 比如 '@typehints(name=str, returns=str)'. 这可能会有用, 但太啰嗦了(增加了一行代码, 并且参数名称必须重复一遍), 且与 PEP 3107 的注解方式相去甚远.
  - 存根文件. 存根文件确实有需要, 但主要是用来把类型提示加入已有代码中, 这些代码本身不适合添加类型提示, 例如: 第三方软件包、需同时支持 Python 2 和 Python 3 的代码、(特别是)拓展模块. 在大多数情况下, 与函数定义放在一起的行内注解会更加有用.
  - 文档字符串. 文档字符串对注解已有约定, 即基于 Sphinx 注解方式 '(:type arg1: description)'. 这真的有点啰嗦(每个参数增加一行代码), 而且不太优雅. 当然再创造一些新语法也是可以的, 但很难超越注解语法(因为它是专为此目的而设计的).
- 还有人提议, 就坐等新的发行版本吧. 但这能解决什么问题呢? 只是会拖延下去而以.

## 十三、PEP 开发过程

- 本 PEP 的最新文稿位于 Github [\[Github\]](#) 上. 另有一个议题跟踪 [\[issues\]](#) 包含了很多技术讨论内容.
- Github 上的文稿会定期小幅更新. 官方 PEP 仓 [\[peps\]](#) (通常)仅在新文稿发布到 python-dev 时才会更新.

## 十四、参考文献

[mypy] <http://mypy-lang.org/>  
[gvr-artima] <https://www.artima.com/weblogs/viewpost.jsp?thread=85551>  
[wiki-variance] [https://en.wikipedia.org/wiki/Covariance\\_and\\_contravariance\\_%28computer\\_science%29](https://en.wikipedia.org/wiki/Covariance_and_contravariance_%28computer_science%29)  
[typeshed] <https://github.com/python/typeshed/>  
[pyflakes] <https://github.com/PyCQA/pyflakes>  
[pylint] <https://www.pylint.org/>  
[roberge] <https://aroberge.blogspot.com/2015/01/type-hinting-in-python-focus-on.html>  
[github] <https://github.com/python/typing>  
[issues] <https://github.com/python/typing/issues>  
[peps] <https://hg.python.org/peps/file/tip/pep-0484.txt>  
[importdocs] <https://docs.python.org/3/reference/import.html#submodules>