PEP 483 -- The Theory of Type Hints

PEP:	483
Title:	The Theory of Type Hints
Author:	Guido van Rossum <guido at python.org>, Ivan Levkivskyi <levkivskyi at gmail.com>
Discussions-To:	Python-Ideas <python-ideas at python.org (mailto:python-ideas@python.org?subject=PEP%20483)>
Status:	Final
Type:	Informational
Created:	19-Dec-2014
Post-History:	
Resolution:	

Contents

Abstract (#id3)

This PEP lays out the theory referenced by PEP 484 (/dev/peps/pep-0484).

Introduction (#id4)

This document lays out the theory of the new type hinting proposal for Python 3.5. It's not quite a full proposal or specification because there are many details that need to be worked out, but it lays out the theory without which it is hard to discuss more detailed specifications. We start by recalling basic concepts of type theory; then we explain gradual typing; then we state some general rules and define the new special types (such as `Union`) that can be used in annotations; and finally we define the approach to generic types and pragmatic aspects of type hinting.

Notational conventions (#id5)

- `t1`, `t2`, etc. and `u1`, `u2`, etc. are types. Sometimes we write `ti` or `tj` to refer to "any of `t1`, `t2`, etc."
- T, U etc. are type variables (defined with `TypeVar()`, see below).
- Objects, classes defined with a class statement, and instances are denoted using standard PEP 8 (/dev/peps/pep-0008) conventions.
- the symbol == applied to types in the context of this PEP means that two expressions represent the same type.
- Note that PEP 484 (/dev/peps/pep-0484) makes a distinction between types and classes (a type is a concept for the type checker, while a class is a runtime concept). In this PEP we clarify this distinction but avoid unnecessary strictness to allow more flexibility in the implementation of type checkers.

Background (#id6)

There are many definitions of the concept of type in the literature. Here we assume that type is a set of values and a set of functions that one can apply to these values.

There are several ways to define a particular type:

- By explicitly listing all values. E.g., `True` and `False` form the type `bool`.
- By specifying functions which can be used with variables of a type. E.g. all objects that have a `__len__` method form the type `Sized`. Both `[1, 2, 3]` and `'abc'` belong to this type, since one can call `len` on them:

```
len([1, 2, 3])  # OK
len('abc')      # also OK
len(42)         # not a member of Sized
```

- By a simple class definition, for example if one defines a class:

```
class UserID(int):
    pass
```

then all instances of this class also form a type.

- There are also more complex types. E.g., one can define the type `FancyList` as all lists containing only instances of `int`, `str` or their subclasses. The value `[1, 'abc', UserID(42)]` has this type.

It is important for the user to be able to define types in a form that can be understood by type checkers. The goal of this PEP is to propose such a systematic way of defining types for type annotations of variables and functions using PEP 3107 (/dev/peps/pep-3107) syntax. These annotations can be used to avoid many kind of bugs, for documentation purposes, or maybe even to increase speed of program execution. Here we only focus on avoiding bugs by using a static type checker.

Subtype relationships (#id7)

A crucial notion for static type checker is the subtype relationship. It arises from the question: If `first_var` has type `first_type`, and `second_var` has type `second_type`, is it safe to assign `first_var = second_var`?

A strong criterion for when it *should* be safe is:

- every value from `second_type` is also in the set of values of `first_type`; and
- every function from `first_type` is also in the set of functions of `second_type`.

The relation defined thus is called a subtype relation.

By this definition:

- Every type is a subtype of itself.
- The set of values becomes smaller in the process of subtyping, while the set of functions becomes larger.

An intuitive example: Every `Dog` is an `Animal`, also `Dog` has more functions, for example it can bark, therefore `Dog` is a subtype of `Animal`. Conversely, `Animal` is not a subtype of `Dog`.

A more formal example: Integers are subtype of real numbers. Indeed, every integer is of course also a real number, and integers support more operations, such as, e.g., bitwise shifts << and >>:

```
lucky_number = 3.14    # type: float
lucky_number = 42      # Safe
lucky_number * 2       # This works
lucky_number << 5      # Fails

unlucky_number = 13    # type: int
unlucky_number << 5    # This works
unlucky_number = 2.72  # Unsafe
```

Let us also consider a tricky example: If `List[int]` denotes the type formed by all lists containing only integer numbers, then it is *not* a subtype of `List[float]`, formed by all lists that contain only real numbers. The first condition of subtyping holds, but appending a real number only works with `List[float]` so that the second condition fails:

```
def append_pi(lst: List[float]) -> None:
    lst += [3.14]

my_list = [1, 3, 5]  # type: List[int]

append_pi(my_list)   # Naively, this should be safe...

my_list[-1] << 5     # ... but this fails
```

There are two widespread approaches to *declare* subtype information to type checker.

In nominal subtyping, the type tree is based on the class tree, i.e., `UserID` is considered a subtype of `int`. This approach should be used under control of the type checker, because in Python one can override attributes in an incompatible way:

```
class Base:
    answer = '42' # type: str

class Derived(Base):
    answer = 5 # should be marked as error by type checker
```

In structural subtyping the subtype relation is deduced from the declared methods, i.e., `UserID` and `int` would be considered the same type. While this may occasionally cause confusion, structural subtyping is considered more flexible. We strive to provide support for both approaches, so that structural information can be used in addition to nominal subtyping.

Summary of gradual typing (#id8)

Gradual typing allows one to annotate only part of a program, thus leverage desirable aspects of both dynamic and static typing.

We define a new relationship, is-consistent-with, which is similar to is-subtype-of, except it is not transitive when the new type `Any` is involved. (Neither relationship is symmetric.) Assigning `a_value` to `a_variable` is OK if the type of `a_value` is consistent with the type of `a_variable`. (Compare this to "... if the type of `a_value` is a subtype of the type of `a_variable`", which states one of the fundamentals of OO programming.) The is-consistent-with relationship is defined by three rules:

- A type `t1` is consistent with a type `t2` if `t1` is a subtype of `t2`. (But not the other way around.)
- `Any` is consistent with every type. (But `Any` is not a subtype of every type.)
- Every type is consistent with `Any`. (But every type is not a subtype of `Any`.)

That's all! See Jeremy Siek's blog post What is Gradual Typing (http://wphomes.soic.indiana.edu/jsiek/what-is-gradual-typing/) for a longer explanation and motivation. `Any` can be considered a type that has all values and all methods. Combined with the definition of subtyping above, this places `Any` partially at the top (it has all values) and bottom (it has all methods) of the type hierarchy. Contrast this to `object` -- it is not consistent with most types (e.g. you can't use an `object()` instance where an `int` is expected). IOW both `Any` and `object` mean "any type is allowed" when used to annotate an argument, but only `Any` can be passed no matter what type is expected (in essence, `Any` declares a fallback to dynamic typing and shuts up complaints from the static checker).

Here's an example showing how these rules work out in practice:

Say we have an `Employee` class, and a subclass `Manager`:

```
class Employee: ...
class Manager(Employee): ...
```

Let's say variable `worker` is declared with type `Employee`:

```
worker = Employee()  # type: Employee
```

Now it's okay to assign a `Manager` instance to `worker` (rule 1):

```
worker = Manager()
```

It's not okay to assign an `Employee` instance to a variable declared with type `Manager`:

```
boss = Manager()  # type: Manager
boss = Employee()  # Fails static check
```

However, suppose we have a variable whose type is `Any`:

```
something = some_func()  # type: Any
```

Now it's okay to assign `something` to `worker` (rule 2):

```
worker = something  # OK
```

Of course it's also okay to assign `worker` to `something` (rule 3), but we didn't need the concept of consistency for that:

```
something = worker  # OK
```

Types vs. Classes (#id9)

In Python, classes are object factories defined by the `class` statement, and returned by the `type(obj)` built-in function. Class is a dynamic, runtime concept.

Type concept is described above, types appear in variable and function type annotations, can be constructed from building blocks described below, and are used by static type checkers.

Every class is a type as discussed above. But it is tricky and error prone to implement a class that exactly represents semantics of a given type, and it is not a goal of PEP 484 (/dev/peps/pep-0484). *The static types described in PEP 484, should not be confused with the runtime classes.* Examples:

- `int` is a class and a type.
- `UserID` is a class and a type.
- `Union[str, int]` is a type but not a proper class:

```
class MyUnion(Union[str, int]): ...  # raises TypeError

Union[str, int]()  # raises TypeError
```

Typing interface is implemented with classes, i.e., at runtime it is possible to evaluate, e.g., `Generic[T].__bases__`. But to emphasize the distinction between classes and types the following general rules apply:

- No types defined below (i.e. `Any`, `Union`, etc.) can be instantiated, an attempt to do so will raise `TypeError`. (But non-abstract subclasses of `Generic` can be.)
- No types defined below can be subclassed, except for `Generic` and classes derived from it.
- All of these will raise `TypeError` if they appear in `isinstance` or `issubclass` (except for unparametrized generics).

Fundamental building blocks (#id10)

- **Any**. Every type is consistent with `Any`; and it is also consistent with every type (see above).
- **Union[t1, t2, ...]**. Types that are subtype of at least one of `t1` etc. are subtypes of this.
  - Unions whose components are all subtypes of `t1` etc. are subtypes of this. Example: `Union[int, str]` is a subtype of `Union[int, float, str]`.
  - The order of the arguments doesn't matter. Example: `Union[int, str] == Union[str, int]`.
  - If `ti` is itself a `Union` the result is flattened. Example: `Union[int, Union[float, str]] == Union[int, float, str]`.
  - If `ti` and `tj` have a subtype relationship, the less specific type survives. Example: `Union[Employee, Manager] == Union[Employee]`.
  - `Union[t1]` returns just `t1`. `Union[]` is illegal, so is `Union[()]`
  - Corollary: `Union[..., object, ...]` returns `object`.
- **Optional[t1]**. Alias for `Union[t1, None]`, i.e. `Union[t1, type(None)]`.
- **Tuple[t1, t2, ..., tn]**. A tuple whose items are instances of `t1`, etc. Example: `Tuple[int, float]` means a tuple of two items, the first is an `int`, the second is a `float`; e.g., `(42, 3.14)`.
  - `Tuple[u1, u2, ..., um]` is a subtype of `Tuple[t1, t2, ..., tn]` if they have the same length `n==m` and each `ui` is a subtype of `ti`.
  - To spell the type of the empty tuple, use `Tuple[()]`.
  - A variadic homogeneous tuple type can be written `Tuple[t1, ...]`. (That's three dots, a literal ellipsis; and yes, that's a valid token in Python's syntax.)

- **Callable[[t1, t2, ..., tn], tr]**. A function with positional argument types `t1` etc., and return type `tr`. The argument list may be empty n==0. There is no way to indicate optional or keyword arguments, nor varargs, but you can say the argument list is entirely unchecked by writing `Callable[..., tr]` (again, a literal ellipsis).

We might add:

- **Intersection[t1, t2, ...]**. Types that are subtype of *each* of `t1`, etc are subtypes of this. (Compare to `Union`, which has *at least one* instead of *each* in its definition.)
  - The order of the arguments doesn't matter. Nested intersections are flattened, e.g. `Intersection[int, Intersection[float, str]] == Intersection[int, float, str]`.
  - An intersection of fewer types is a supertype of an intersection of more types, e.g. `Intersection[int, str]` is a supertype of `Intersection[int, float, str]`.
  - An intersection of one argument is just that argument, e.g. `Intersection[int]` is `int`.
  - When argument have a subtype relationship, the more specific type survives, e.g. `Intersection[str, Employee, Manager]` is `Intersection[str, Manager]`.
  - `Intersection[]` is illegal, so is `Intersection[()]`.
  - Corollary: `Any` disappears from the argument list, e.g. `Intersection[int, str, Any] == Intersection[int, str]`. `Intersection[Any, object]` is `object`.
  - The interaction between `Intersection` and `Union` is complex but should be no surprise if you understand the interaction between intersections and unions of regular sets (note that sets of types can be infinite in size, since there is no limit on the number of new subclasses).

Generic types (#id11)

The fundamental building blocks defined above allow to construct new types in a generic manner. For example, `Tuple` can take a concrete type `float` and make a concrete type `Vector = Tuple[float, ...]`, or it can take another type `UserID` and make another concrete type `Registry = Tuple[UserID, ...]`. Such semantics is known as generic type constructor, it is similar to semantics of functions, but a function takes a value and returns a value, while generic type constructor takes a type and "returns" a type.

It is common when a particular class or a function behaves in such a type generic manner. Consider two examples:

- Container classes, such as `list` or `dict`, typically contain only values of a particular type. Therefore, a user might want to type annotate them as such:

```
users = [] # type: List[UserID]
users.append(UserID(42)) # OK
users.append('Some guy') # Should be rejected by the type checker

examples = {} # type: Dict[str, Any]
examples['first example'] = object() # OK
examples[2] = None                   # rejected by the type checker
```

- The following function can take two arguments of type `int` and return an `int`, or take two arguments of type `float` and return a `float`, etc.:

```
def add(x, y):
    return x + y

add(1, 2) == 3
add('1', '2') == '12'
add(2.7, 3.5) == 6.2
```

To allow type annotations in situations from the first example, built-in containers and container abstract base classes are extended with type parameters, so that they behave as generic type constructors. Classes, that behave as generic type constructors are called *generic types*. Example:

```
from typing import Iterable

class Task:
    ...

def work(todo_list: Iterable[Task]) -> None:
    ...
```

Here `Iterable` is a generic type that takes a concrete type `Task` and returns a concrete type `Iterable[Task]`.

Functions that behave in the type generic manner (as in second example) are called *generic functions*. Type annotations of generic functions are allowed by *type variables*. Their semantics with respect to generic types is somewhat similar to semantics of parameters in functions. But one does not assign concrete types to type variables, it is the task of a static type checker to find their possible values and warn the user if it cannot find. Example:

```
def take_first(seq: Sequence[T]) -> T: # a generic function
    return seq[0]

accumulator = 0 # type: int

accumulator += take_first([1, 2, 3])   # Safe, T deduced to be int
accumulator += take_first((2.7, 3.5))  # Unsafe
```

Type variables are used extensively in type annotations, also internal machinery of the type inference in type checkers is typically build on type variables. Therefore, let us consider them in detail.

Type variables (#id12)

`X = TypeVar('X')` declares a unique type variable. The name must match the variable name. By default, a type variable ranges over all possible types. Example:

```
def do_nothing(one_arg: T, other_arg: T) -> None:
    pass

do_nothing(1, 2)               # OK, T is int
do_nothing('abc', UserID(42))  # also OK, T is object
```

`Y = TypeVar('Y', t1, t2, ...)`. Ditto, constrained to `t1`, etc. Behaves similar to `Union[t1, t2, ...]`. A constrained type variable ranges only over constrains `t1`, etc. *exactly*; subclasses of the constrains are replaced by the most-derived base class among `t1`, etc. Examples:

- Function type annotation with a constrained type variable:

```
S = TypeVar('S', str, bytes)

def longest(first: S, second: S) -> S:
    return first if len(first) >= len(second) else second

result = longest('a', 'abc')  # The inferred type for result is str

result = longest('a', b'abc')  # Fails static type check
```

  In this example, both arguments to `longest()` must have the same type (`str` or `bytes`), and moreover, even if the arguments are instances of a common `str` subclass, the return type is still `str`, not that subclass (see next example).

- For comparison, if the type variable was unconstrained, the common subclass would be chosen as the return type, e.g.:

```
S = TypeVar('S')

def longest(first: S, second: S) -> S:
    return first if len(first) >= len(second) else second

class MyStr(str): ...

result = longest(MyStr('a'), MyStr('abc'))
```

  The inferred type of `result` is `MyStr` (whereas in the `AnyStr` example it would be `str`).

- Also for comparison, if a `Union` is used, the return type also has to be a `Union`:

```
U = Union[str, bytes]


def longest(first: U, second: U) -> U:
    return first if len(first) >= len(second) else second


result = longest('a', 'abc')
```

The inferred type of `result` is still `Union[str, bytes]`, even though both arguments are `str`.

Note that the type checker will reject this function:

```
def concat(first: U, second: U) -> U:
    return x + y  # Error: can't concatenate str and bytes
```

For such cases where parameters could change their types only simultaneously one should use constrained type variables.

Defining and using generic types (#id13)

Users can declare their classes as generic types using the special building block `Generic`. The definition `class MyGeneric(Generic[X, Y, ...]): ...` defines a generic type `MyGeneric` over type variables `X`, etc. `MyGeneric` itself becomes parameterizable, e.g. `MyGeneric[int, str, ...]` is a specific type with substitutions `X -> int`, etc. Example:

```
class CustomQueue(Generic[T]):

    def put(self, task: T) -> None:
        ...
    def get(self) -> T:
        ...

def communicate(queue: CustomQueue[str]) -> Optional[str]:
    ...
```

Classes that derive from generic types become generic. A class can subclass multiple generic types. However, classes derived from specific types returned by generics are not generic. Examples:

```
class TodoList(Iterable[T], Container[T]):
    def check(self, item: T) -> None:
        ...

def check_all(todo: TodoList[T]) -> None:  # TodoList is generic
    ...

class URLList(Iterable[bytes]):
    def scrape_all(self) -> None:
        ...

def search(urls: URLList) -> Optional[bytes]  # URLList is not generic
    ...
```

Subclassing a generic type imposes the subtype relation on the corresponding specific types, so that `TodoList[t1]` is a subtype of `Iterable[t1]` in the above example.

Generic types can be specialized (indexed) in several steps. Every type variable could be substituted by a specific type or by another generic type. If `Generic` appears in the base class list, then it should contain all type variables, and the order of type parameters is determined by the order in which they appear in `Generic`. Examples:

```
Table = Dict[int, T]     # Table is generic
Messages = Table[bytes]  # Same as Dict[int, bytes]

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

class DerivedGeneric(BaseGeneric[int, T]): # DerivedGeneric has one parameter
    ...

SpecificType = DerivedGeneric[int]         # OK

class MyDictView(Generic[S, T, U], Iterable[Tuple[U, T]]):
    ...

Example = MyDictView[list, int, str]       # S -> list, T -> int, U -> str
```

If a generic type appears in a type annotation with a type variable omitted, it is assumed to be `Any`. Such form could be used as a fallback to dynamic typing and is allowed for use with `issubclass` and `isinstance`. All type information in instances is erased at runtime. Examples:

```
def count(seq: Sequence) -> int:      # Same as Sequence[Any]
    ...

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

class UserClass:
    ...

issubclass(UserClass, FrameworkBase)  # This is OK

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

IntNode = Node[int]
my_node = IntNode()  # at runtime my_node.__class__ is Node
                     # inferred static type of my_node is Node[int]
```

Covariance and Contravariance (#id14)

If `t2` is a subtype of `t1`, then a generic type constructor `GenType` is called:

- Covariant, if `GenType[t2]` is a subtype of `GenType[t1]` for all such `t1` and `t2`.
- Contravariant, if `GenType[t1]` is a subtype of `GenType[t2]` for all such `t1` and `t2`.
- Invariant, if neither of the above is true.

To better understand this definition, let us make an analogy with ordinary functions. Assume that we have:

```
def cov(x: float) -> float:
    return 2*x

def contra(x: float) -> float:
    return -x

def inv(x: float) -> float:
    return x*x
```

If `x1 < x2`, then *always* `cov(x1) < cov(x2)`, and `contra(x2) < contra(x1)`, while nothing could be said about `inv`. Replacing `<` with is-subtype-of, and functions with generic type constructor we get examples of covariant, contravariant, and invariant behavior. Let us now consider practical examples:

- `Union` behaves covariantly in all its arguments. Indeed, as discussed above, `Union[t1, t2, ...]` is a subtype of `Union[u1,` u2 ...] if t1 is a subtype of u1, etc.

- `FrozenSet[T]` is also covariant. Let us consider `int` and `float` in place of T. First, `int` is a subtype of `float`. Second, set of values of `FrozenSet[int]` is clearly a subset of values of `FrozenSet[float]`, while set of functions from `FrozenSet[float]` is a subset of set of functions from `FrozenSet[int]`. Therefore, by definition `FrozenSet[int]` is a subtype of `FrozenSet[float]`.
- `List[T]` is invariant. Indeed, although set of values of `List[int]` is a subset of values of `List[float]`, only `int` could be appended to a `List[int]`, as discussed in section "Background". Therefore, `List[int]` is not a subtype of `List[float]`. This is a typical situation with mutable types, they are typically invariant.

One of the best examples to illustrate (somewhat counterintuitive) contravariant behavior is the callable type. It is covariant in the return type, but contravariant in the arguments. For two callable types that differ only in the return type, the subtype relationship for the callable types follows that of the return types. Examples:

- `Callable[[], int]` is a subtype of `Callable[[], float]`.
- `Callable[[], Manager]` is a subtype of `Callable[[], Employee]`.

While for two callable types that differ only in the type of one argument, the subtype relationship for the callable types goes *in the opposite direction* as for the argument types. Examples:

- `Callable[[float], None]` is a subtype of `Callable[[int], None]`.
- `Callable[[Employee], None]` is a subtype of `Callable[[Manager], None]`.

Yes, you read that right. Indeed, if a function that can calculate the salary for a manager is expected:

```
def calculate_all(lst: List[Manager], salary: Callable[[Manager], Decimal]):
    ...
```

then `Callable[[Employee], Decimal]` that can calculate a salary for any employee is also acceptable.

The example with `Callable` shows how to make more precise type annotations for functions: choose the most general type for every argument, and the most specific type for the return value.

It is possible to *declare* the variance for user defined generic types by using special keywords `covariant` and `contravariant` in the definition of type variables used as parameters. Types are invariant by default. Examples:

```
T = TypeVar('T')
T_co = TypeVar('T_co', covariant=True)
T_contra = TypeVar('T_contra', contravariant=True)

class LinkedList(Generic[T]):  # invariant by default
    ...
    def append(self, element: T) -> None:
        ...

class Box(Generic[T_co]):      #  this type is declared covariant
    def __init__(self, content: T_co) -> None:
        self._content = content
    def get_content(self) -> T_co:
        return self._content

class Sink(Generic[T_contra]): # this type is declared contravariant
    def send_to_nowhere(self, data: T_contra) -> None:
        with open(os.devnull, 'w') as devnull:
            print(data, file=devnull)
```

Note, that although the variance is defined via type variables, it is not a property of type variables, but a property of generic types. In complex definitions of derived generics, variance *only* determined from type variables used. A complex example:

```
T_co = TypeVar('T_co', Employee, Manager, covariant=True)
T_contra = TypeVar('T_contra', Employee, Manager, contravariant=True)

class Base(Generic[T_contra]):
    ...

class Derived(Base[T_co]):
    ...
```

A type checker finds from the second declaration that `Derived[Manager]` is a subtype of `Derived[Employee]`, and `Derived[t1]` is a subtype of `Base[t1]`. If we denote the is-subtype-of relationship with <, then the full diagram of subtyping for this case will be:

```
Base[Manager]    >  Base[Employee]
    v                   v
Derived[Manager] <  Derived[Employee]
```


so that a type checker will also find that, e.g., `Derived[Manager]` is a subtype of `Base[Employee]`.

For more information on type variables, generic types, and variance, see PEP 484 (https://www.python.org/dev/peps/pep-0484), the mypy docs on generics (http://mypy.readthedocs.io/en/latest/generics.html), and Wikipedia (http://en.wikipedia.org/wiki/Covariance_and_contravariance_%28computer_science%29).

Pragmatics (#id15)

Some things are irrelevant to the theory but make practical use more convenient. (This is not a full list; I probably missed a few and some are still controversial or not fully specified.)

- Where a type is expected, `None` can be substituted for `type(None)`; e.g. `Union[t1, None] == Union[t1, type(None)]`.
- Type aliases, e.g.:

```
Point = Tuple[float, float]
def distance(point: Point) -> float: ...
```

- Forward references via strings, e.g.:

```
class MyComparable:
    def compare(self, other: 'MyComparable') -> int: ...
```

- If a default of `None` is specified, the type is implicitly `Optional`, e.g.:

```
def get(key: KT, default: VT = None) -> VT: ...
```

- Type variables can be declared in unconstrained, constrained, or bounded form. The variance of a generic type can also be indicated using a type variable declared with special keyword arguments, thus avoiding any special syntax, e.g.:

```
T = TypeVar('T', bound=complex)

def add(x: T, y: T) -> T:
    return x + y

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

class ImmutableList(Generic[T_co]): ...
```

- Type declaration in comments, e.g.:

```
lst = []  # type: Sequence[int]
```

```
zork = cast(Any, frobozz())
```

- Other things, e.g. overloading and stub modules, see PEP 484 (/dev/peps/pep-0484).

Predefined generic types and Protocols in typing.py (#id16)

(See also the typing.py module (https://github.com/python/typing/blob/master/src/typing.py).)

- Everything from `collections.abc` (but `Set` renamed to `AbstractSet`).
- `Dict`, `List`, `Set`, `FrozenSet`, a few more.
- `re.Pattern[AnyStr]`, `re.Match[AnyStr]`.
- `io.IO[AnyStr]`, `io.TextIO ~ io.IO[str]`, `io.BinaryIO ~ io.IO[bytes]`.

Copyright (#id17)

This document is licensed under the Open Publication License (http://www.opencontent.org/openpub/) [1] (#id1).

References and Footnotes (#id18)

[1] (#id2) http://www.opencontent.org/openpub/ (http://www.opencontent.org/openpub/)

Source: https://github.com/python/peps/blob/master/pep-0483.txt (https://github.com/python/peps/blob/master/pep-0483.txt)

Tweets by @ThePSF

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

Back to Top

Copyright ©2001-2019. Python Software Foundation Legal Statements Privacy Policy Powered by Heroku