Type Hints

Python 3.x 3.5–3.6

Examples

Adding types to a function

Let's take an example of a function which receives two arguments and returns a value indicating their sum:

```
def two_sum(a, b):
    return a + b
```

By looking at this code, one can not safely and without doubt indicate the type of the arguments for function two_sum . It works both when supplied with int values:

```
print(two_sum(2, 1))  # result: 3
```

and with strings:

```
print(two_sum("a", "b"))  # result: "ab"
```

and with other values, such as list s, tuple s et cetera.

Due to this dynamic nature of python types, where many are applicable for a given operation, any type checker would not be able to reasonably assert whether a call for this function should be allowed or not.

To assist our type checker we can now provide type hints for it in the Function definition indicating the type that we allow.

To indicate that we only want to allow int types we can change our function definition to look like:

```
def two_sum(a: int, b: int):
    return a + b
```

Annotations follow the argument name and are separated by a : character.

Similarly, to indicate only str types are allowed, we'd change our function to specify it:

```
def two_sum(a: str, b: str):
    return a + b
```

Apart from specifying the type of the arguments, one could also indicate the return value of a function call. This is done by adding the -> character followed by the type after the closing parenthesis in the argument list *but* before the : at the end of the function declaration:

```
def two_sum(a: int, b: int) -> int:
    return a + b
```

Now we've indicated that the return value when calling two_sum should be of type int . Similarly we can define appropriate values for str , float , list , set and others.

Although type hints are mostly used by type checkers and IDEs, sometimes you may need to retrieve them. This can be done using the __annotations__ special attribute:

```
two_sum.__annotations__
# {'a': <class 'int'>, 'b': <class 'int'>, 'return': <class 'int'>}
```

Generic Types

The typing.TypeVar is a generic type factory. It's primary goal is to serve as a parameter/placeholder for generic function/class/method annotations:

```
import typing

T = typing.TypeVar("T")

def get_first_element(l: typing.Sequence[T]) -> T:
    """Gets the first element of a sequence."""
    return l[0]
```

NamedTuple

Creating a namedtuple with type hints is done using the function NamedTuple from the typing module:

```
import typing
Point = typing.NamedTuple('Point', [('x', int), ('y', int)])
```

Note that the name of the resulting type is the first argument to the function, but it should be assigned to a variable with the same name to ease the work of type checkers.

Variables and Attributes

Variables are annotated using comments:

```
x = 3  # type: int
x = negate(x)
x = 'a type-checker might catch this error'
```

Python 3.x ≥3.6

Starting from Python 3.6, there is also new syntax for variable annotations . The code above might use the form

```
x: int = 3
```

Unlike with comments, it is also possible to just add a type hint to a variable that was not previously declared, without setting a value to it:

```
y: int
```

Additionally if these are used in the module or the class level, the type hints can be retrieved using typing.get_type_hints(class_or_module) :

```
class Foo:
    x: int
    y: str = 'abc'

print(typing.get_type_hints(Foo))
# ChainMap({'x': <class 'int'>, 'y': <class 'str'>}, {})
```

Alternatively, they can be accessed by using the __annotations__ special variable or attribute:

```
x: int
print(__annotations__)
# {'x': <class 'int'>}

class C:
    s: str
print(C.__annotations__)
# {'s': <class 'str'>}
```

Class Members and Methods

```
class A:
    x = None  # type: float
    def __init__(self, x: float) -> None:
        """
        self should not be annotated
        init should be annotated to return None
        """
        self.x = x

    @classmethod
    def from_int(cls, x: int) -> 'A':
        """
        cls should not be annotated
        Use forward reference to refer to current class with string literal 'A'
        """
        return cls(float(x))
```

Forward reference of the current class is needed since annotations are evaluated when the function is defined. Forward references can also be used when referring to a class that would cause a circular import if imported.

Type hints for keyword arguments

```
def hello_world(greeting: str = 'Hello'):
    print(greeting + ' world!')
```

Note the spaces around the equal sign as opposed to how keyword arguments are usually styled.

Syntax

```
typing.Callable[[int, str], None] -> def func(a: int, b: str) -> None
```

```
typing.Mapping[str, int] -> {"a": 1, "b": 2, "c": 3}
```

```
typing.List[int] -> [1, 2, 3]
```

```
typing.Set[int] -> {1, 2, 3}
```

```
typing.Optional[int] -> None or int
```

```
typing.Sequence[int] -> [1, 2, 3] or (1, 2, 3)
```

```
typing.Any -> Any type
```

```
typing.Union[int, str] -> 1 or "1"
```

```
T = typing.TypeVar('T') -> Generic type
```

Parameters

Remarks

Type Hinting, as specified in PEP 484 , is a formalized solution to statically indicate the type of a value for Python Code. By appearing alongside the `typing` module, type-hints offer Python users the capability to annotate their code thereby assisting type checkers while, indirectly, documenting their code with more information.