**typing**

下面我们再来详细看下 typing 模块的具体用法，这里主要会介绍一些常用的注解类型，如 List、Tuple、Dict、Sequence 等等，了解了每个类型的具体使用方法，我们可以得心应手的对任何变量进行声明了。

在引入的时候就直接通过 typing 模块引入就好了，例如：

|  |  |
| --- | --- |
| 1 | from typing import List, Tuple |

**List**

List、列表，是 list 的泛型，基本等同于 list，其后紧跟一个方括号，里面代表了构成这个列表的元素类型，如由数字构成的列表可以声明为：

|  |  |
| --- | --- |
| 1 | var: List[int or float] = [2, 3.5] |

另外还可以嵌套声明都是可以的：

|  |  |
| --- | --- |
| 1 | var: List[List[int]] = [[1, 2], [2, 3]] |

**Tuple、NamedTuple**

Tuple、元组，是 tuple 的泛型，其后紧跟一个方括号，方括号中按照顺序声明了构成本元组的元素类型，如 Tuple[X, Y] 代表了构成元组的第一个元素是 X 类型，第二个元素是 Y 类型。

比如想声明一个元组，分别代表姓名、年龄、身高，三个数据类型分别为 str、int、float，那么可以这么声明：

|  |  |
| --- | --- |
| 1 | person: Tuple[str, int, float] = ('Mike', 22, 1.75) |

同样地也可以使用类型嵌套。

NamedTuple，是 collections.namedtuple 的泛型，实际上就和 namedtuple 用法完全一致，但个人其实并不推荐使用 NamedTuple，推荐使用 attrs 这个库来声明一些具有表征意义的类。

**Dict、Mapping、MutableMapping**

Dict、字典，是 dict 的泛型；Mapping，映射，是 collections.abc.Mapping 的泛型。根据官方文档，Dict 推荐用于注解返回类型，Mapping 推荐用于注解参数。它们的使用方法都是一样的，其后跟一个中括号，中括号内分别声明键名、键值的类型，如：

|  |  |
| --- | --- |
| 1  2 | def size(rect: Mapping[str, int]) -> Dict[str, int]:   return {'width': rect['width'] + 100, 'height': rect['width'] + 100} |

这里将 Dict 用作了返回值类型注解，将 Mapping 用作了参数类型注解。

MutableMapping 则是 Mapping 对象的子类，在很多库中也经常用 MutableMapping 来代替 Mapping。

**Set、AbstractSet**

Set、集合，是 set 的泛型；AbstractSet、是 collections.abc.Set 的泛型。根据官方文档，Set 推荐用于注解返回类型，AbstractSet 用于注解参数。它们的使用方法都是一样的，其后跟一个中括号，里面声明集合中元素的类型，如：

|  |  |
| --- | --- |
| 1  2 | def describe(s: AbstractSet[int]) -> Set[int]:   return set(s) |

这里将 Set 用作了返回值类型注解，将 AbstractSet 用作了参数类型注解。

**Sequence**

Sequence，是 collections.abc.Sequence 的泛型，在某些情况下，我们可能并不需要严格区分一个变量或参数到底是列表 list 类型还是元组 tuple 类型，我们可以使用一个更为泛化的类型，叫做 Sequence，其用法类似于 List，如：

|  |  |
| --- | --- |
| 1  2 | def square(elements: Sequence[float]) -> List[float]:   return [x \*\* 2 for x in elements] |

**NoReturn**

NoReturn，当一个方法没有返回结果时，为了注解它的返回类型，我们可以将其注解为 NoReturn，例如：

|  |  |
| --- | --- |
| 1  2 | def hello() -> NoReturn:   print('hello') |

**Any**

Any，是一种特殊的类型，它可以代表所有类型，静态类型检查器的所有类型都与 Any 类型兼容，所有的无参数类型注解和返回类型注解的都会默认使用 Any 类型，也就是说，下面两个方法的声明是完全等价的：

|  |  |
| --- | --- |
| 1  2  3  4  5 | def add(a):   return a + 1    def add(a: Any) -> Any:   return a + 1 |

原理类似于 object，所有的类型都是 object 的子类。但如果我们将参数声明为 object 类型，静态参数类型检查便会抛出错误，而 Any 则不会，具体可以参考官方文档的说明：[https://docs.python.org/zh-cn/3/library/typing.html?highlight=typing#the-any-type](https://docs.python.org/zh-cn/3/library/typing.html?highlight=typing" \l "the-any-type" \t "_blank)。

**TypeVar**

TypeVar，我们可以借助它来自定义兼容特定类型的变量，比如有的变量声明为 int、float、None 都是符合要求的，实际就是代表任意的数字或者空内容都可以，其他的类型则不可以，比如列表 list、字典 dict 等等，像这样的情况，我们可以使用 TypeVar 来表示。

例如一个人的身高，便可以使用 int 或 float 或 None 来表示，但不能用 dict 来表示，所以可以这么声明：

|  |  |
| --- | --- |
| 1  2  3  4 | height = 1.75  Height = TypeVar('Height', int, float, None)  def get\_height() -> Height:   return height |

这里我们使用 TypeVar 声明了一个 Height 类型，然后将其用于注解方法的返回结果。

**NewType**

NewType，我们可以借助于它来声明一些具有特殊含义的类型，例如像 Tuple 的例子一样，我们需要将它表示为 Person，即一个人的含义，但但从表面上声明为 Tuple 并不直观，所以我们可以使用 NewType 为其声明一个类型，如：

|  |  |
| --- | --- |
| 1  2 | Person = NewType('Person', Tuple[str, int, float])  person = Person(('Mike', 22, 1.75)) |

这里实际上 person 就是一个 tuple 类型，我们可以对其像 tuple 一样正常操作。

**Callable**

Callable，可调用类型，它通常用来注解一个方法，比如我们刚才声明了一个 add 方法，它就是一个 Callable 类型：

|  |  |
| --- | --- |
| 1 | print(Callable, type(add), isinstance(add, Callable)) |

运行结果：

typing.Callable <class 'function'> True

在这里虽然二者 add 利用 type 方法得到的结果是 function，但实际上利用 isinstance 方法判断确实是 True。

Callable 在声明的时候需要使用 Callable[[Arg1Type, Arg2Type, ...], ReturnType] 这样的类型注解，将参数类型和返回值类型都要注解出来，例如：

|  |  |
| --- | --- |
| 1  2  3  4  5 | def date(year: int, month: int, day: int) -> str:   return f'{year}-{month}-{day}'    def get\_date\_fn() -> Callable[[int, int, int], str]:   return date |

这里首先声明了一个方法 date，接收三个 int 参数，返回一个 str 结果，get\_date\_fn 方法返回了这个方法本身，它的返回值类型就可以标记为 Callable，中括号内分别标记了返回的方法的参数类型和返回值类型。

**Union**

Union，联合类型，Union[X, Y] 代表要么是 X 类型，要么是 Y 类型。

联合类型的联合类型等价于展平后的类型：

|  |  |
| --- | --- |
| 1 | Union[Union[int, str], float] == Union[int, str, float] |

仅有一个参数的联合类型会坍缩成参数自身，比如：

|  |  |
| --- | --- |
| 1 | Union[int] == int |

多余的参数会被跳过，比如：

|  |  |
| --- | --- |
| 1 | Union[int, str, int] == Union[int, str] |

在比较联合类型的时候，参数顺序会被忽略，比如：

|  |  |
| --- | --- |
| 1 | Union[int, str] == Union[str, int] |

这个在一些方法参数声明的时候比较有用，比如一个方法，要么传一个字符串表示的方法名，要么直接把方法传过来：

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | def process(fn: Union[str, Callable]):   if isinstance(fn, str):    # str2fn and process    pass   elif isinstance(fn, Callable):    fn() |

这样的声明在一些类库方法定义的时候十分常见。

**Optional**

Optional，意思是说这个参数可以为空或已经声明的类型，即 Optional[X] 等价于 Union[X, None]。

但值得注意的是，这个并不等价于可选参数，当它作为参数类型注解的时候，不代表这个参数可以不传递了，而是说这个参数可以传为 None。

如当一个方法执行结果，如果执行完毕就不返回错误信息， 如果发生问题就返回错误信息，则可以这么声明：

|  |  |
| --- | --- |
| 1  2 | def judge(result: bool) -> Optional[str]:   if result: return 'Error Occurred' |

**Generator**

如果想代表一个生成器类型，可以使用 Generator，它的声明比较特殊，其后的中括号紧跟着三个参数，分别代表 YieldType、SendType、ReturnType，如：

|  |  |
| --- | --- |
| 1  2  3  4  5 | def echo\_round() -> Generator[int, float, str]:   sent = yield 0   while sent >= 0:    sent = yield round(sent)   return 'Done' |

在这里 yield 关键字后面紧跟的变量的类型就是 YieldType，yield 返回的结果的类型就是 SendType，最后生成器 return 的内容就是 ReturnType。

当然很多情况下，生成器往往只需要 yield 内容就够了，我们是不需要 SendType 和 ReturnType 的，可以将其设置为空，如：

|  |  |
| --- | --- |
| 1  2  3  4 | def infinite\_stream(start: int) -> Generator[int, None, None]:   while True:    yield start    start += 1 |