第15章. 高级话题

当你阅读到本章时，恭喜你，你已经学会了Python语法中所有最实用的部分。本章讨论的内容是Python语法中不那么实用的部分，至少在99%的情况下没有必要使用它们。因此在深入讨论Python的这些高级话题之前，有必要向你介绍这段话：

> **Beautiful is better than ugly.（优美好过丑陋。）**
> **Explicit is better than implicit.（清晰好过晦涩。）**
> **Simple is better than complex.（简单好过复杂。）**
> **Flat is better than nested.（扁平好过嵌套。）**
> **Sparse is better than dense.（松散好过紧凑。）**
> **Readability counts.（可读性是必要的。）**
> **Special cases aren't special enough to break the rules.（在规则面前没有特例。）**
> **Although practicality beats purity.（然而实用性比教条主义更重要。）**
> **Errors should never pass silently.（错误不应被默然放过。）**
> **Unless explicitly silenced.（除非有意如此。）**
> **In the face of ambiguity, refuse the temptation to guess.（当面对模棱两可时，拒绝盲目猜测。）**
> **There should be one-- and preferably only one --obvious way to do it.（任何事都存在一种最好的做法。）**
> **Although that way may not be obvious at first unless you're Dutch.（虽然该做法起初可能只有Python之父知道。）**
> **Now is better than never.（动手好过不做。）**
> **Although never is often better than *right* now.（然而不做好过不假思索就动手。）**
> **If the implementation is hard to explain, it's a bad idea.（如果一个实现很难向人解释，那肯定不是一个好主意。）**
> **If the implementation is easy to explain, it may be a good idea.（如果一个实现很容易向人解释，那可能是一个好主意。）**
> **Namespaces are one honking great idea -- let's do more of those!（名字空间是一种绝妙的理念，让我们多一些这样的想法。）**

这段话被称为“Python之禅（Zen of Python）”，是由Python和CPython的主要贡献者之一蒂姆·彼得斯（Tim Peters）提出的，被PEP 20描述，并作为彩蛋嵌入了CPython。在启动CPython后第一次执行“import this”时就会看到它。它告诫我们，除非确实是“最好的做法”，否则没必要为了炫技而使用本章介绍的技巧。

15-1. 元类

（语言参考手册：3.3.3、3.3.11）
（标准库：types）

第3章已经初步说明了什么是元类，并提到了__prepare__和__slots__。同时它还提到了__bases__和__mro__。第5章在讨论用数据描述器实现封装时提到了__set_name__，在讨论继承时解释了什么是“方法解析序列（MRO）”。在这一节，我们将把这些知识串联起来，形成定义类的完整图景。

让我们重新来审视类定义语句的语法：

```
class clsname[(argument_list)]:
    suite
```

其中argument_list代表的类参数列表中，所有基于位置对应的参数都被视为基类，而所有基于关键字对应的参数则对该类进行某方面的设置，特别的metaclass关键字用于指定该类所属的元类。为了便于讨论，类定义语句的语法在本节被写为：

```
class clsname[(base1, base2, ..., metaclass=metacls, **kwds)]:
    suite
```

下面详细讨论该类定义语句的执行步骤。

步骤一是解析MRO条目。这是指通过types模块定义的resolve_bases()函数来对指定的基类进行处理，其语法为：

```
types.resolve_bases(bases)
```

其中bases参数需被传入格式为“(base1, base2, ...)”的元组。该函数会返回一个元组，包含处理过的基类，处理规则为：

- 如果baseN是元类type的实例，则不做任何处理。
- 如果baseN不是元类type的实例，且不具有__mro_entries__属性，则不做任何处理。
- 如果baseN不是元类type的实例，但具有__mro_entries__属性，则调用该属性，用其返回的类对象序列替代baseN。__mro_entries__属性接收通过bases参数传入的元组而非仅baseN自身，并返回一个类对象形成的元组。如果它返回空元组，则意味着baseN被忽略。

types.resolve_bases()返回的元组将被新建类对象的__bases__特殊属性引用。

步骤二是确定元类。确定一个类的元类将按照如下规则：

➤ 如果省略了metaclass关键字，且没有指定基类，则以type作为元类。
➤ 如果通过metaclass关键字指定了一个自定义类，且它不是type的实例，则直接以该自定义类作为元类。
➤ 如果通过metaclass关键字指定了一个type的实例，或者指定了基类，则使用“最近派生元类（most derived metaclass）”。

最近派生元类是这样确定的：以通过metaclass关键字指定的类和所有指定基类的元类（即type(base1)、type(base2)、……）为候选元类，从中找出一个同时是所有候选元类的子类的元类（一个类可以视为其本身的子类）。如果不存在最近派生元类，则抛出TypeError异常，意味着发生了“元类冲突”。被确定的元类将被新建类对象的__class__特殊属性引用。

步骤三是确定MRO。按照第5章介绍的C3方法计算出MRO。最后得到的元组将被新建类对象的__mro__特殊属性引用。

步骤四是准备类名字空间。如果元类具有__prepare__属性，则会以如下形式调用它：

```
namespace = metacls.__prepare__(name, bases, **kwds)
```

其中name为新建类的类名，bases为__bases__引用的元组，而kwds是metaclass之外的用于设置类的关键字。该函数返回一个映射。如果元类不具有__prepare__，则将使用一个空的有序映射。

步骤五是执行类体。这类似于调用了：

```
exec(body, globals(), namespace)
```

其中body是对应类体的代码对象，namespace需传入通过步骤四得到的名字空间。唯一的区别是当类定义语句位于函数体内时，允许类体访问函数体中定义的本地变量（参考6-4节给出的例子）。执行类体的结果是填充了名字空间。

步骤六是创建类对象。这相当于调用了：

```
metacls(name, bases, dic, **kwds)
```

其中dic是通过步骤四得到的名字空间，其余参数的含义与在__prepare__中相同。注意第5章介绍的type()的第二种语法其实只是一个特例，所有元类都必须支持这种形式的调用。这会导致元类的__new__和__init__被先后调用，并被传入上面的参数列表，而__new__的返回值就是新建类。在第3章中曾经简单提到了这一步骤，这里再重申如下：__new__会创建一个空字典作为新建类的变量字典，而__init__会用名字空间dic填充该变量字典。这意味着

不论名字空间是哪种类型的映射，新建类的变量字典都必然是字典。然而新建类的__dict__会引用该变量字典的只读代理，而非该变量字典本身，这就保证了类对象被创建后其实例属性不能被修改。

以上6个步骤是创建任何类时都需要执行的。而当以type或者从type派生出的类作为元类时，在类对象被创建之后，还会执行如下额外两个步骤。

步骤七，遍历新建类的所有类属性，如果该属性引用的对象具有__set_name__魔术属性，则以如下方式调用它：

```
clsname.attribute.__set_name__(clsname, attribute)
```

这是第5章讨论数据描述器时encapsulation2.py中的代码有效的关键。

步骤八，遍历新建类的所有直接基类（即__bases__引用的元组中的类对象），如果某个类对象具有__init_subclass__魔术属性，则以如下方式调用它：

```
baseN.__init_subclass__(cls, **kwds)
```

其中cls是新建类，kwds是metaclass之外的用于设置类的关键字。

在详细了解一个类的创建过程后，我们就可以通过重写该过程中用到的属性来对类的创建过程进行更精细的控制。这些控制有些是通过自定义基类实现的，有些是通过自定义元类实现的，但不论是哪种情况，都会影响到一大批类。

当我们想以自定义类为元类时，首先要确保该类没有__slots__，或者__slots__指定的标识符中具有__dict__，因为所有类对象（包括类型对象）都需要具有变量字典。其次需要考虑该自定义类是否以type作为基类。如果不从type派生元类，那么我们需要考虑实现表3-6中的哪些属性（其中__call__是必须实现的）。此外，如果该自定义类是type的实例（注意type派生的元类的实例必然是type的实例），就会导致使用最近派生元类。而在这种情况下若还指定了基类，则这些基类必须都以该自定义类或其派生类为元类，否则会导致元类冲突。事实上，由于自行实现__call__是相当麻烦的，所以在实践中我们几乎总是以type派生的类为自定义元类。

有时候最近派生元类很难一眼看出来，此时可以使用types模块提供的如下辅助函数：

```
types.prepare_class(name, bases=(), kwds=None)
```

其中name参数需传入新建类的类名，bases需被传入其直接基类列表，kwds需被传入包含所有关键字（包括metaclass）的映射。而该函数会返回格式为“(metaclass, namespace, kwds)”的元组，其中metaclass引用最近派生元类，namespace引用由该元类生成的变量字典，而kwds则去掉metaclass之后剩下的关键字。

下面的例子说明了最近派生元类的选择方法：

```
import types


#定义第一个元类。
class meta1(type):
    pass


#定义第二个元类。
class meta2(type):
    pass


#定义第三个元类。
class meta3(meta1, meta2):
    pass


#定义属于meta1的类。
class A(metaclass=meta1):
    pass


#定义属于meta2的类。
class B(metaclass=meta2):
    pass


#定义属于meta3的类。
class C(metaclass=meta3):
    pass


#由于A的元类是meta1，所以最近派生元类是meta1。
print(types.prepare_class("O", (A,), {"metaclass": meta1}))

#由于A的元类是meta1，meta2和meta1之间没有继承关系，所以发生元类冲突。
try:
    print(types.prepare_class("O", (A,), {"metaclass": meta2}))
except TypeError as e:
    print(repr(e))

#由于A的元类是meta1，而meta3是meta1的子类，所以最近派生元类是meta3。
print(types.prepare_class("O", (A,), {"metaclass": meta3}))

#由于A的元类是meta1，B的元类是meta2，所以发生元类冲突。
try:
    print(types.prepare_class("O", (A, B)))
except TypeError as e:
    print(repr(e))

#由于A的元类是meta1，C的元类是meta3，而meta3是meta1的子类，所以最近派生元类
# 是meta3。
print(types.prepare_class("O", (A, C)))
```

```
    #由于B的元类是meta2，C的元类是meta3，而meta3是meta2的子类，所以最近派生元类
    # 是meta3。
    print(types.prepare_class("O", (B, C)))

    #由于A的元类是meta1，B的元类是meta2，meta3同时是meta1和meta2的子类，所以最
    # 近派生元类是meta3。
    print(types.prepare_class("O", (A, B), {"metaclass": meta3}))

    #下面三条语句说明选取最近派生元类的算法是依次两两比较。

    #首先比较A的元类meta1和B的元类meta2，发生元类冲突，不再继续比较。
    try:
        print(types.prepare_class("O", (A, B, C)))
    except TypeError as e:
        print(repr(e))

    #首先比较A的元类meta1和C的元类meta3，选中meta3，然后再比较meta3和B的元类meta2，
    # 最后依然选中meta3。
    print(types.prepare_class("O", (A, C, B)))

    #首先比较C的元类meta3和A的元类meta1，选中meta3，然后再比较meta3和B的元类meta2，
    # 最后依然选中meta3。
    print(types.prepare_class("O", (C, A, B)))

    #下面的语句说明通过metaclass关键字指定的元类将与第一个基类比较，且是首先被比较，因
    # 此发生元类冲突。
    try:
        print(types.prepare_class("O", (A, C, B), {"metaclass": meta2}))
    except TypeError as e:
        print(repr(e))
```

请将上述代码保存为metaclass1.py，然后通过如下命令行验证：

```
    $ python3 metaclass1.py
    (<class '__main__.meta1'>, {}, {})
    TypeError('metaclass conflict: the metaclass of a derived class must be a
(non-strict) subclass of the metaclasses of all its bases')
    (<class '__main__.meta3'>, {}, {})
    TypeError('metaclass conflict: the metaclass of a derived class must be a
(non-strict) subclass of the metaclasses of all its bases')
    (<class '__main__.meta3'>, {}, {})
    (<class '__main__.meta3'>, {}, {})
    (<class '__main__.meta3'>, {}, {})
    TypeError('metaclass conflict: the metaclass of a derived class must be a
(non-strict) subclass of the metaclasses of all its bases')
    (<class '__main__.meta3'>, {}, {})
    (<class '__main__.meta3'>, {}, {})
    TypeError('metaclass conflict: the metaclass of a derived class must be a
(non-strict) subclass of the metaclasses of all its bases')
```

注意上面的例子说明了计算最近派生元类的算法是按照如下方式运行的：首先根据直接基类的元类生成一个序列（保持直接基类的顺序），然后将通过metaclass关键字（如果有的话）指定的元类放在该序列的开头，最后按照起泡排序算法中的方法进行依次两两比较。这意味着，仅当最近派生元类是该序列的前两个元素之一时，才能100%的保证它会被找出，否则即便存在合法的最近派生元类依然有可能发生元类冲突。所以最稳妥的做法是先手工利用types.prepare_class()找出最近派生元类，然后通过metaclass关键字直接指定它。这也是types.prepare_class()的意义所在。

自定义元类的主要目的是通过重写__prepare__、__new__和__init__来影响属于它的类的创建过程，以及通过重写__call__来影响属于它的类被实例化时的行为。利用这些接口能实现的功能是千变万化的。下面举几个例子。

第一个例子通过重写__prepare__使得属于该元类的类的所有不以“_”开头的类属性名都会被自动添加“clsname_”形式的前缀，其中“clsname”代表类名：

```
#自定义映射。
class PrefixMapping(dict):
    def __init__(self, prefix):
        self.prefix = prefix

    def __setitem__(self, key, value):
        if key[0] != "_":
            dict.__setitem__(self, self.prefix + "_" + key, value)
        else:
            dict.__setitem__(self, key, value)


#定义元类。
class PrefixMeta(type):
    @classmethod
    def __prepare__(cls, name, bases, **kwds):
        return PrefixMapping(name)


#定义属于该元类的第一个类。
class A(metaclass=PrefixMeta):
    dimension = 2

    def __init__(self, x, y):
        self.x = x
        self.y = y

    def move(self, dx, dy):
        self.x += dx
        self.y += dy

    def location(self):
        print("(" + str(self.x) + ", " + str(self.y) + ")")


#定义属于该元类的第二个类。
class B(metaclass=PrefixMeta):
    dimension = 2

    def __init__(self, x, y):
        self.x = x
        self.y = y

    def move(self, dx, dy):
        self.x += dx
        self.y += dy

    def location(self):
        print("(" + str(self.x) + ", " + str(self.y) + ")")
```

请将上述代码保存为metaclass2.py，然后通过如下命令行和语句验证：

```
      $ python3 -i metaclass2.py

      >>> a = A(15, 37)
      >>> a.x
      15
      >>> a.y
      37
      >>> a.A_move(-1, 1)
      >>> a.A_location()
      (14, 38)
      >>> A.A_dimension
      2
      >>> A.__dict__
      mappingproxy({'__module__': '__main__', 'A_dimension': 2, '__init__':
<function A.__init__ at 0x1091a72e0>, 'A_move': <function A.move at 0x1091a7380>,
'A_location': <function A.location at 0x1091a7420>, '__dict__': <attribute
'__dict__' of 'A' objects>, '__weakref__': <attribute '__weakref__' of 'A'
objects>, '__doc__': None})
      >>> A.n = 0
      >>> A.n
      0
      >>>
      >>> b = B(-9, -12)
      >>> b.x
      -9
      >>> b.y
      -12
      >>> b.B_move(2, -2)
      >>> b.B_location()
      (-7, -14)
      >>> B.B_dimension
      2
      >>> B.__dict__
      mappingproxy({'__module__': '__main__', 'B_dimension': 2, '__init__':
<function B.__init__ at 0x1091a74c0>, 'B_move': <function B.move at 0x1091a7560>,
'B_location': <function B.location at 0x1091a7600>, '__dict__': <attribute
'__dict__' of 'B' objects>, '__weakref__': <attribute '__weakref__' of 'B'
objects>, '__doc__': None})
      >>> B.m = 1
      >>> B.m
      1
      >>>
```

从该例子可以看出，虽然类A和类B的类体完全相同（都拷贝自Point2D.py中类Point2D的类体），但最后A的类属性中dimension变成了A_dimension、move()变成了A_move()，location()变成了A_location()；而B的类属性中dimension变成了B_dimension、move()变成了B_move()，location()变成了B_location()。两者以“_”开头的类属性，创建后被添加的类属性，以及它们实例的实例属性都不受影响。这是怎么做到的呢？

我们首先自定义了一个映射类PrefixMapping，它派生自dict，但__init__被重写以记录下指定的前缀，__setitem__也被重写使得当像该映射插入键值对时，会给不以“_”开头的键添加指定的前缀。然后我们自定义了元类PrefixMeta，并重写__prepare__，使它返回一个PrefixMapping对象作为命名空间，而在创建该对象时传入新建类的类名为参数，使得以“clsname_”为前缀。这样当A或B的类体被执行时，每条赋值语句或函数定义语句是在给该PrefixMapping对象插入键值对，因而不以“_”开头的类属性会被添加类名前缀。而由于该类被创建后，PrefixMapping对象已经不复存在，其内的键值对已经被拷贝到了变量字典

中，所以之后添加的类属性（例如A.n和B.m）不会被添加类名前缀。类被实例化时实例属性是在__init__中设置的，因此也不会被添加类名前缀。

最后需要强调，__prepare__必须被实现为一个类方法。

第二个例子通过重写__new__，使得属于该元类的类的所有非函数类属性的不以“_”开头的属性名都自动大写：

```
import types


#定义元类。
class ConstantMeta(type):
    @staticmethod
    def __new__(cls, name, bases, dic, **kwds):
        new_dic = {}
        for k, v in dic.items():
            if isinstance(v, types.FunctionType) or k[0] == '_':
                new_dic[k] = v
            else:
                new_dic[k.upper()] = v
        return type.__new__(cls, name, bases, new_dic, **kwds)


#定义属于该元类的类。
class Point(metaclass=ConstantMeta):
    dimension = 2

    def __init__(self, x, y):
        self.x = x
        self.y = y

    def move(self, dx, dy):
        self.x += dx
        self.y += dy

    def location(self):
        print("(" + str(self.x) + ", " + str(self.y) + ")")
```

请将上述代码保存为metaclass3.py，然后通过如下命令行和语句验证：

```
$ python3 -i metaclass3.py

>>> Point.__dict__
mappingproxy({'__module__': '__main__', 'DIMENSION': 2, '__init__':
<function Point.__init__ at 0x1046af2e0>, 'move': <function Point.move at
0x1046af380>, 'location': <function Point.location at 0x1046af420>, '__dict__':
<attribute '__dict__' of 'Point' objects>, '__weakref__': <attribute
'__weakref__' of 'Point' objects>, '__doc__': None})
>>>
```

该结果说明Point类的dimension属性被自动改写为DIMENSION。这是因为ConstantMeta的__new__遍历了名字空间中的所有键值对，将值不是函数而又不以“_”开头的键进行了大写。

第三个例子通过重写__init__和__call__，使得属于该元类的类只能有一个实例：

```
#定义元类。
class Singleton(type):
    #为属于该元类的每个类都添加类属性instance。
    def __init__(self, *args, **kwargs):
        self.instance = None
        super().__init__(*args, **kwargs)

    #检查属于该元类的类的instance属性，仅当它不引用None时才允许实例化。
    def __call__(self, *args, **kwargs):
        if self.instance is None:
            self.instance = super().__call__(*args, **kwargs)
        return self.instance


#定义第一个属于Singleton的类。
class A(metaclass=Singleton):
    def __init__(self, n, *args):
        self.a = n
        super().__init__(*args)


#定义第二个属于Singleton的类。
class B(metaclass=Singleton):
    def __init__(self, n, *args):
        self.b = n
        super().__init__(*args)


#定义第三个属于Singleton的类。   它同时以A和B为直接基类。
class C(A, B):
    def __init__(self, n, m, l):
        self.c = l
        super().__init__(n, m)
```

请将上述代码保存为metaclass4.py，然后通过如下命令行和语句验证类A、类B和类C都只能创建一个实例：

```
$ python3 -i metaclass4.py

>>> A.instance
>>> a1 = A(0)
>>> A.instance
<__main__.A object at 0x10f72ff90>
>>> a2 = A(1)
>>> A.instance
<__main__.A object at 0x10f72ff90>
>>> a1 is a2
True
>>> a1.a
0
>>> a2.a
0
>>>
>>> B.instance
>>> b1 = B(3)
```

```
>>> B.instance
<__main__.B object at 0x10f72ffd0>
>>> b2 = B(4)
>>> B.instance
<__main__.B object at 0x10f72ffd0>
>>> b1 is b2
True
>>> b1.b
3
>>> b2.b
3
>>>
>>> C.instance
>>> c1 = C(10, 20, 30)
>>> C.instance
<__main__.C object at 0x10f7400d0>
>>> c2 = C(40, 50, 60)
>>> C.instance
<__main__.C object at 0x10f7400d0>
>>> c1 is c2
True
>>> (c1.a, c1.b, c1.c)
(10, 20, 30)
>>> (c2.a, c2.b, c2.c)
(10, 20, 30)
>>> a1 is b1
False
>>> a1 is c1
False
>>> b1 is c1
False
>>>
```

该例子的关键在于，通过__init__给属于Singleton的每个类都增加了一个类属性instance，而通过__call__使得实例化这些类时先检查instance属性是否引用None，如果不是则直接返回它引用的对象。

想要影响类创建过程也不一定要自定义元类。我们也可以给基类添加__mro_entries__和/或__init__subclass__，以及给类属性引用的对象所属类添加__set_name__。广义上讲，这些也属于元类编程。__mro_entries__用于支持泛型，将在后面讨论。__set_name__的用法已经在第5章的例子encapsulation2.py中进行了演示。下面的例子说明了如何使用__init_subclass__，它使得一个类无法被作为基类：

```
class A():
    #@classmethod
    def __init_subclass__(cls):
        raise Exception("class A cannot be used as a base class!")

class B(A):
    pass
```

请将上述代码保存为metaclass5.py，然后通过如下命令行验证：

```
$ python3 metaclass5.py
Traceback (most recent call last):
  File "/Users/wwy/metaclass5.py", line 6, in <module>
    class B(A):
    ^^^^^^^^^^^
  File "/Users/wwy/metaclass5.py", line 4, in __init_subclass__
    raise Exception("class A cannot be used as a base class!")
    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
Exception: class A cannot be used as a base class!
```

需要强调的是，__init_subclass__也必须被实现为一个类方法，但添加@classmethod却不是必须的，即使省略了该装饰器__init_subclass__也会自动称为类方法。

上面给出的例子都比较简单，下面给出一个复杂一些的例子，它结合使用了自定义元类、反射机制和__init__subclass__，使得属于它的类的类属性是固定的，既不能增加也不能减少。这些类需要在类定义语句中通过关键字attributes来指定自己具有哪些类属性：

```
#定义元类。
class FixedAttrMeta(type):
    def __init__(self, *args, **kwargs):
        #从类定义语句中的attributes关键字获得类属性及其初始值。   将其记录到attrs
        # 属性中。
        self.attrs = kwargs["attributes"]
        #初始化所有类属性。
        for k, v in self.attrs.items():
            setattr(self, k, v)
        super().__init__(*args, **kwargs)

    #通过反射使得不能添加类属性，也不能修改attrs属性。
    def __setattr__(self, name, value):
        if name == "attrs":
            if not hasattr(self, "attrs"):
                super().__setattr__(name, value)
            else:
                raise Exception("Can not modify attribute attrs!")
        elif name in self.attrs:
            super().__setattr__(name, value)
        else:
            raise Exception(f"Can not add attribute {name}!")

    #通过反射使得不能删除类属性。
    def __delattr__(self, name):
        if name == "attrs" or name in self.attrs:
            raise Exception(f"Can not delete attribute {name}!")
        else:
            return None


#定义一个基类，使得__init_subclass__可以接受任何参数。
class FixedAttrBase():
    @classmethod
    def __init_subclass__(cls, *args, **kwargs):
        object.__init_subclass__()


#定义属于FixedAttrMeta的类A，它以FixedAttrBase为基类，除了attr外还具有x和y两个
# 类属性。
```

```
    class A(FixedAttrBase, metaclass=FixedAttrMeta, attributes={"x": 0,
"y":0}):
        pass
```

请将上述代码保存为metaclass6.py，然后通过如下命令行和语句验证：

```
    $ python3 -i metaclass6.py

    >>> A.__dict__
    mappingproxy({'__module__': '__main__', '__doc__': None, 'attrs': {'x': 0,
'y': 0}, 'x': 0, 'y': 0})
    >>> A.attrs = {}
    Traceback (most recent call last):
      File "<stdin>", line 1, in <module>
      File "/Users/wwy/metaclass6.py", line 18, in __setattr__
        raise Exception("Can not modify attribute attrs!")
        ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
    Exception: Can not modify attribute attrs!
    >>> del A.attrs
    Traceback (most recent call last):
      File "<stdin>", line 1, in <module>
      File "/Users/wwy/metaclass6.py", line 27, in __delattr__
        raise Exception(f"Can not delete attribute {name}!")
        ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
    Exception: Can not delete attribute attrs!
    >>> A.x = 12
    >>> A.x
    12
    >>> del A.x
    Traceback (most recent call last):
      File "<stdin>", line 1, in <module>
      File "/Users/wwy/metaclass6.py", line 27, in __delattr__
        raise Exception(f"Can not delete attribute {name}!")
        ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
    Exception: Can not delete attribute x!
    >>> A.y = -10
    >>> A.y
    -10
    >>> del A.y
    Traceback (most recent call last):
      File "<stdin>", line 1, in <module>
      File "/Users/wwy/metaclass6.py", line 27, in __delattr__
        raise Exception(f"Can not delete attribute {name}!")
        ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
    Exception: Can not delete attribute y!
    >>> A.z = 0
    Traceback (most recent call last):
      File "<stdin>", line 1, in <module>
      File "/Users/wwy/metaclass6.py", line 22, in __setattr__
        raise Exception(f"Can not add attribute {name}!")
        ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
    Exception: Can not add attribute z!
    >>>
```

注意该例子中的基类FixedAttrBase是必须的，这是因为A的类定义语句使用了自定义关键字attributes，而metaclass之外的关键字会作为参数传递给A的基类的__init_subclass__。由于object类实现了__init__subclass__，但它什么也不做，仅在接收到任何参数时抛出异

常，所以如果A不以FixedAttrBase为基类，就会调用object类的__init__subclass__，导致抛出异常。

在看过上面的诸多元类编程的例子后，相信你已经对如何使用元类有了基本的认知。本书不再对此深入讨论。蒂姆·彼得斯除了Python之禅外还有下面一段话：

> “Metaclasses are deeper magic than 99% of users should ever worry about. If you wonder whether you need them, you don't (the people who actually need them know with certainty that they need them, and don't need an explanation about why).”
> （“元类是深度的魔法，99%的用户根本不应为此操心。如果你不知道你是否需要它们，那么你就不需要它们（实际需要它们的人非常清楚它们的用处，因此不需要解释为什么需要它们）。”）

这段话概括起来，就是“不要滥用元类”。事实上，只有Python专家们需要掌握元类，他们会在开发类似于Django这样的框架时广泛使用元类，以实现一些Python本身不具有的语法特征。

15-2. 抽象基类

（语言参考手册：3.3.4、3.3.7、3.3.8）
（标准库：collections.abc、numbers、abc）

第5章提到了内置函数issubclass()和isinstance()，说明了它们分别用于判断一个类是否是另一个类的子类，以及一个对象是否是一个类的实例。然而第5章并没有说明这两个内置函数是如何实现的，也没有说明如果它们返回True，到底意味着什么。

事实上，issubclass()是依赖于元类中的__subclasscheck__属性实现的，后者语法为：

```
cls.__subclasscheck__(self, subclass)
```

其subclass参数需被传入一个类，不能被传入类对象形成的元组或类的联合类型。如果该函数返回True，则说明subclass应被视为cls的子类；否则返回False。

类似的，isinstance()是依赖于元类中的__instancecheck__属性实现的，后者语法为：

```
cls.__instancecheck__(self, instance)
```

其instance参数需被传入一个对象。如果该函数返回True，则说明instance应被视为cls的实例；否则返回True。

表3-6说明，元类type已经实现了__subclasscheck__和__instancecheck__。它们都通过遍历MRO来实现各自的功能的。这意味着默认情况下，当__subclasscheck__返回True时，subclass确实是cls的子类；当__instancecheck__返回True时，instance确实是cls的实例。

然而，人们很快发现，有些情况下需要将一个类视为另一个类的子类，哪怕它并不真正是该类的子类；有些情况下需要将一个对象视为一个类的实例，哪怕它并不真正是该类的实例。举例来说，如果一个对象具有下列属性：

- __len__：支持返回该对象容纳的元素的个数。
- __iter__：支持迭代。
- __reversed__：支持反向迭代。
- __contains__：支持in和not in运算。
- __getitem__：支持抽取和切片。
- count()：支持返回指定元素出现的次数。
- index()：支持返回指定元素首次出现的位置。

那么哪怕该对象所属类没有以任意一种内置序列类型或标准库的collections模块中定义的序列类型为基类，也应被判断为是一种序列。这种判断的意义在于使我们可以确定能对该对象执行任何能对序列执行的操作。

为了解决上述问题，PEP 3119引入了“抽象基类（abstract base classes，ABC）”这一概念，并给标准库增加了abc模块作为该技术的实现。简单来说，抽象基类就是拥有至少一个“抽象方法（abstract methods）”的类。（从Python 3.3开始，抽象方法不再区分方法、类方法和静态方法。）抽象基类不能被实例化，但可以被普通的类继承，也可以相互继承。然而类继承抽象基类时，必须重写抽象基类中的所有抽象方法。否则它就不是类，而依然是抽象基类。

上面的规则保证了永远不可能通过实例调用一个抽象方法。但可以通过抽象基类本身调用抽象方法，也可以在类对其进行重写时通过super()调用抽象方法。这意味着抽象方法也需要具有函数体，而这是Python的抽象基类与其他编程语言（例如Java）的抽象基类不同的地方。

抽象基类也可以包含不属于抽象方法的方法、类方法和静态方法，它们被统称为“混入方法（mixin methods）”。混入方法的访问规则与普通的类中的方法、类方法和静态方法相同。事实上，抽象基类甚至可以继承普通的类，而后者的所有方法都自动成为前者的混入方法。

抽象基类的意义在于，如果一个类被视为一个抽象基类的子类，或一个对象被视为一个抽象基类的实例，那就可以放心地通过该类或该对象访问该抽象基类具有的所有抽象方法和混入方法，这使我们可以据此判断能够对该类或该对象执行哪些操作。

下面让我们看看abc模块提供了哪些接口使我们可以按照上面的描述定义和使用抽象基类。首先，abc.ABCMeta是所有抽象基类的元类，在定义抽象基类时需通过metaclass关键

字指定使用它。此外，abc.ABC是一个未定义任何属性的类，它以abc.ABCMeta为元类，因此如果以abc.ABC为基类，则也会导致以abc.ABCMeta为元类。在多继承的情况下，必须考虑发生元类冲突的可能，但abc.ABCMeta派生自type，所以只要通过metaclass关键字指定使用abc.ABCMeta作为元类，或者将以abc.ABCMeta或其子类为元类的类作为第一个直接基类，就可以保证符合抽象基类的要求。

仅以abc.ABCMeta或其子类为元类并不意味着创建的是抽象基类。仅当一个类具有至少一个抽象方法时才是抽象基类。下面的例子说明了这点：

```
$ python3

>>> import abc
>>> class FakeABC(metaclass=abc.ABCMeta):
...     pass
...
>>> obj = FakeABC()
>>>
```

注意FakeABC以abc.ABCMeta为元类，但不具有任何属性，所以仍然能够被实例化，证明它不是抽象基类。

定义抽象方法必须使用abc模块提供的abstractmethod装饰器，而该装饰器仅能在以abc.ABCMeta或其子类为元类的类中使用。如果将abstractmethod装饰器与其他装饰器联合使用，则应以abstractmethod作为最内层的装饰器。下面的例子说明了如何定义和使用抽象基类：

```
import abc


#定义类A。
class A:
    #该方法会成为继承A的抽象基类的混入方法。
    def a_mixin(self):
        print("a_mixin")


#定义抽象基类B。
class B(metaclass=abc.ABCMeta):
    #该方法是B的抽象方法。
    @abc.abstractmethod
    def b(self):
        print("b")


#定义抽象基类C。
class C(abc.ABC):
    #该类方法是C的抽象方法。
    @classmethod
    @abc.abstractmethod
    def c(cls):
        print("c")

    #该方法是C的混入方法。
    def c_mixin(self):
```

```
        print("c_mixin")


#定义抽象基类D。   它继承了类A，因此还具有混入方法a_mixin()。
class D(A, metaclass=abc.ABCMeta):
    #该静态方法是D的抽象方法。
    @staticmethod
    @abc.abstractmethod
    def d():
        print("d")


#定义抽象基类E。   它继承了抽象基类B，因此还具有抽象方法b()。   它也继承了抽象基类C，
# 因此还具有抽象方法c()和混入方法c_mixin()。
class E(B, C):
    #该方法是E的混入方法。
    def e_mixin(self):
        print("e_mixin")


#定义类F，它继承了抽象基类D。
class F(D):
    #重写D的抽象方法d()。
    @staticmethod
    def d():
        print("F's implementation of d()")
        #通过super()调用D的d()。   这是通过super()访问基类的静态方法的例子。
        super(F, F).d()


#定义类G，它继承了抽象基类E。
class G(E):
    #重写B的抽象方法b()。
    def b(self):
        print("G's implementation of b()")
        #通过super()调用B的b()。   这是通过super()访问基类的方法的例子。
        super().b()

    #重写C的抽象方法c()。
    @classmethod
    def c(cls):
        print("G's implementation of c()")
        #通过super()调用C的c()。   这是通过super()访问基类的类方法的例子。
        super().c()
```

这个例子定义的类和抽象基类之间的继承关系如图15-1所示（绿色斜体代表抽象基类，下同）：

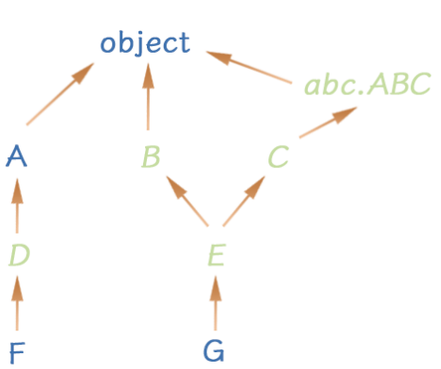

图15-1. 类和抽象基类之间的继承关系

请将上述代码保存为abc1.py，然后通过如下命令行和语句验证：

```
$ python3 -i abc1.py

>>> o1 = F()
>>> o1.a_mixin()
a_mixin
>>> o1.d()
F's implementation of d()
d
>>> o2 = G()
>>> o2.b()
G's implementation of b()
b
>>> o2.c()
G's implementation of c()
c
>>> o2.c_mixin()
c_mixin
>>>
```

abc.ABCMeta重写了__subclasscheck__和__instancecheck__，为它们增加了一些功能，但当这些额外功能不适用时，它们会分别自动调用type对它们的实现。换句话说，当一个类继承了一个抽象基类，那么它也会被视为该抽象基类的子类，它的实例也会被视为该抽象基类的实例。请继续前面的例子，通过如下语句验证：

```
>>> issubclass(F, D)
True
>>> issubclass(F, A)
True
>>> isinstance(o1, D)
True
>>> isinstance(o1, A)
True
>>> issubclass(G, E)
True
>>> issubclass(G, B)
True
>>> issubclass(G, C)
True
>>> isinstance(o2, E)
True
>>> isinstance(o2, B)
True
>>> isinstance(o2, C)
True
>>>
```

那么abc.ABCMeta重写的__subclasscheck__和__instancecheck__增加了哪些额外功能呢？第一个额外功能是使得每个抽象基类都可以通过调用register()将一个并没有以它为基类的类注册为它的“虚子类（virtual subclasses）”，进而使该类能够被视为它的子类，该类的实例能够被视为它的实例。

register()是通过abc.ABCMeta实现的，可以被作为一个类装饰器来使用，语法为：

```
@abc.ABCMeta.register(subclass)
```

当然，直接调用该函数也是可以的。不论哪种情况，它都会返回传入subclass参数的类。

register()的真正作用是把虚子类与抽象基类的映射关系记录到abc.ABCMeta的缓冲区中，而这会导致描述该缓冲区状态令牌发生变化。abc模块提供了get_cache_token()使我们可以取得该令牌，其语法为：

```
abc.get_cache_token()
```

该函数返回的令牌本质上是一个整数，因此支持相等性测试。

请继续前面的例子，通过如下语句验证register()和get_cache_token()的用法：

```
>>> class H():
...     pass
...
>>> B.register(H)
<class '__main__.H'>
>>> issubclass(H, B)
True
>>> o3 = H()
>>> isinstance(o3, B)
True
>>> o3.b()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'H' object has no attribute 'b'
>>> abc.get_cache_token()
25
>>>
>>> class I():
...     @classmethod
...     def c(cls):
...         print("I's implementation of c()")
...
>>> C.register(I)
<class '__main__.I'>
>>> issubclass(I, C)
True
>>> o4 = I()
>>> o4.c()
I's implementation of c()
>>> o4.c_mixin()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'I' object has no attribute 'c_mixin'
>>> abc.get_cache_token()
26
>>>
```

从该结果可以看出，我们可以把任何类注册为一个抽象基类的虚子类。滥用register()的危害也是显而易见的：如果一个虚子类没有实现该抽象基类的所有抽象方法（例如上面的H没有实现B的b()），或者没有实现该抽象基类的所有混入方法（例如上面的I没有实现C的c_mixin()），就会违背抽象基类的使用规则。

请继续通过如下语句验证：

```
>>> class J():
...     pass
...
>>> D.register(J)
<class '__main__.J'>
>>> issubclass(J, D)
True
>>> issubclass(J, A)
False
>>> J.__mro__
(<class '__main__.J'>, <class 'object'>)
>>>
>>> class K():
...     pass
...
>>> E.register(K)
<class '__main__.K'>
>>> issubclass(K, E)
True
>>> issubclass(K, B)
True
>>> issubclass(K, C)
True
>>> issubclass(K, abc.ABC)
True
>>> K.__mro__
(<class '__main__.K'>, <class 'object'>)
>>>
```

从该结果可以看出，将一个类注册为某个抽象基类的虚子类（例如J注册为D的虚子类，K注册为E的虚子类），那么它也就自动被注册为该抽象基类的所有抽象基类的虚子类（例如K是B、C和abc.ABC的虚子类），但它不会成为该抽象基类的某个非抽象的基类的子类（例如J不是A的子类）。

下面给出一个合理使用register()的例子。该例子自定义了一种数据类型——二元整型向量，并定义了它们之间的各种运算，然后将其注册为numbers.Complex的虚子类。在给出该例子的具体代码之前，首先有必要介绍一下Python标准库中的number模块，它提供了用于实现Python中数字的抽象基类，其层次结构如图15-2所示。而表15-1则总结了所有这些抽象基类引入的抽象方法和混入方法（注意其中的函数和运算符应被理解为相应的魔术属性，例如“+”代表__pos__、__add__和_radd__）。

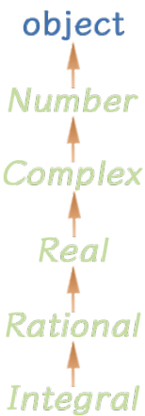

图15-2. numbers的抽象基类

表15-1. numbers中的抽象基类的接口

抽象基类	抽象方法	混入方法
Number	无	无
Complex	bool()、complex()、real()、imag()、conjugage()、abs()、+、*、/、**、==	-、!=
Real	float()、math.trunc()、math.floor()、math.ceil()、round()、divmod()、//、%、<、<=、>、>=	无
Rational	numerator、denominator	无
Integral	int()、pow()、<<、>>、&、^、|、~	无

如果我们自定义一个类，将其注册为numbers中的某个抽象基类的虚子类，并实现所有的抽象方法和混入方法，那么该类就可以被视为一个数字类。然而可被视为一个数字类，并不意味着应直接继承相应的抽象基类，因为该类实现抽象方法和混入方法的方式很可能并不兼容内置的数字类型。在这种情况下，使用register()要合理的多。本例子中的二元整型向量就符合这样的情况：

```
import abc
import numbers
import math


#定义代表二元整型向量的类。
class BiIntVector():
    #定义一个取得二元整型向量实部的描述器。
    class Real:
        def __get__(self, instance, owner=None):
            if instance:
                return instance.x
            else:
                return self

    #定义一个取得二元整型向量虚部的描述器。
    class Imag:
        def __get__(self, instance, owner=None):
            if instance:
                return instance.y
```

```
            else:
                return self

    #初始化二元整型向量。
    def __init__(self, x, y):
        x = int(x)
        y = int(y)
        self.x = x
        self.y = y

    #将二元整型向量转换为字符串。
    def __str__(self):
        return "(" + str(self.x) + ", " + str(self.y) + ")"

    def __repr__(self):
        return "(" + str(self.x) + ", " + str(self.y) + ")"

    #取得二元整型向量的实部。
    real = Real()

    #取得二元整型向量的虚部。
    imag = Imag()

    #计算共轭二元整型向量，即将虚部设置为其相反数。
    def conjugate(self):
        return BiIntVector(self.x, -self.y)

    #将二元整型向量转换为复数，该复数的实部和虚部分别等于二元整型向量的实部和虚部。
    def __complex__(self):
        return complex(self.x, self.y)

    #计算二元整型向量的绝对值，即该向量的长度。
    def __abs__(self):
        return math.sqrt(self.x**2 + self.y**2)

    #对二元整型向量进行逻辑值检测，仅当其绝对值为0时才返回False，其他情况返回True。
    def __bool__(self):
        if self.__abs__() == 0:
            return False
        else:
            return True

    #实现二元整型向量之间的==和!=比较。
    def __eq__(self, other):
        if isinstance(other, BiIntVector):
            return self.x == other.x and self.y == other.y
        else:
            return NotImplemented

    def __ne__(self, other):
        if isinstance(other, BiIntVector):
            return self.x != other.x or self.y != other.y
        else:
            return NotImplemented

    #实现二元整型向量的加法，注意操作数必须都是二元整型向量。
    def __pos__(self):
        return self

    def __add__(self, other):
        if isinstance(other, BiIntVector):
            return BiIntVector(self.x + other.x, self.y + other.y)
        else:
            return NotImplemented
```

```
    def __radd__(self, other):
        if isinstance(other, BiIntVector):
            return BiIntVector(other.x + self.x, other.y + self.y)
        else:
            return NotImplemented

    def __iadd__(self, other):
        print("augmented addition!")
        return self.__add__(other)

    #实现二元整型向量的减法，注意操作数必须都是二元整型向量。
    def __neg__(self):
        return BiIntVector(-self.x, -self.y)

    def __sub__(self, other):
        if isinstance(other, BiIntVector):
            return BiIntVector(self.x - other.x, self.y - other.y)
        else:
            return NotImplemented

    def __rsub__(self, other):
        if isinstance(other, BiIntVector):
            return BiIntVector(other.x - self.x, other.y - self.y)
        else:
            return NotImplemented

    def __isub__(self, other):
        print("augmented subtraction!")
        return self.__sub__(other)

    #实现二元整型向量的乘法，注意操作数必须都是二元整型向量。
    def __mul__(self, other):
        if isinstance(other, BiIntVector):
            return BiIntVector(self.x * other.x, self.y * other.y)
        else:
            return NotImplemented

    def __rmul__(self, other):
        if isinstance(other, BiIntVector):
            return BiIntVector(other.x * self.x, other.y * self.y)
        else:
            return NotImplemented

    def __imul__(self, other):
        print("augmented multiplication!")
        return self.__mul__(other)

    #实现二元整型向量的除法，注意操作数必须都是二元整型向量。
    def __truediv__(self, other):
        if isinstance(other, BiIntVector):
            return BiIntVector(self.x // other.x, self.y // other.y)
        else:
            return NotImplemented

    def __rtruediv__(self, other):
        if isinstance(other, BiIntVector):
            return BiIntVector(other.x // self.x, other.y // self.y)
        else:
            return NotImplemented

    def __itruediv__(self, other):
        print("augmented division!")
        return self.__truediv__(other)

    #实现二元整型向量的乘方，注意操作数必须都是二元整型向量。
```

```
        def __pow__(self, other):
            if isinstance(other, BiIntVector):
                return BiIntVector(self.x ** other.x, self.y ** other.y)
            else:
                return NotImplemented

        def __rpow__(self, other):
            if isinstance(other, BiIntVector):
                return BiIntVector(other.x ** self.x, other.y ** self.y)
            else:
                return NotImplemented

        def __ipow__(self, other):
            print("augmented power!")
            return self.__pow__(other)


    #将BiIntVector注册为Complex的虚子类。
    numbers.Complex.register(BiIntVector)
```

请将上述代码保存为abc2.py，接下类一步一步验证这个例子的合理性。

首先，通过如下命令行和语句验证BiIntVector是Complex和Number的虚子类，但不是Real、Rational和Integral的虚子类：

```
$ python3 -i abc2.py

>>> issubclass(BiIntVector, numbers.Number)
True
>>> issubclass(BiIntVector, numbers.Complex)
True
>>> issubclass(BiIntVector, numbers.Real)
False
>>> issubclass(BiIntVector, numbers.Rational)
False
>>> issubclass(BiIntVector, numbers.Integral)
False
>>>
```

下面的语句验证了BiIntVector支持实部、虚部和共轭，以及它能够转化为复数：

```
>>> v1 = BiIntVector(3, 4)
>>> v1
(3, 4)
>>> v1.real
3
>>> v1.imag
4
>>> v1.conjugate()
(3, -4)
>>> v2 = v1.conjugate()
>>> complex(v1)
(3+4j)
>>> complex(v2)
(3-4j)
>>>
```

下面的语句验证了BiIntVector的绝对值与逻辑值检测之间的关系，以及二元整型向量之间的相等和不等关系：

```
>>> v0 = BiIntVector(0, 0)
>>> abs(v0)
0.0
>>> abs(v1)
5.0
>>> abs(v2)
5.0
>>> bool(v0)
False
>>> bool(v1)
True
>>> bool(v2)
True
>>> v1 == v2
False
>>> v1 != v2
True
>>>
```

下面的语句验证了BiIntVector对正号、加法和基于加法的增强赋值运算的支持，注意虽然BiIntVector对象可以转化为复数，但不支持与复数相加：

```
>>> +v1
(3, 4)
>>> v1 + v2
(6, 0)
>>> v3 = v2
>>> v3
(3, -4)
>>> v3 += v1
augmented addition!
>>> v3
(6, 0)
>>> v3 + v0
(6, 0)
>>> v3 + 0j
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'BiIntVector' and 'complex'
>>>
```

下面的语句验证了BiIntVector对负号、减法和基于减法的增强赋值运算的支持，注意虽然BiIntVector对象可以转化为复数，但不支持与复数相减：

```
>>> -v1
(-3, -4)
>>> v1 - v2
(0, 8)
>>> v4 = v2
>>> v4
(3, -4)
```

```
>>> v4 -= v1
augmented subtraction!
>>> v4
(0, -8)
>>> v4 - v0
(0, -8)
>>> v4 - 0j
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for -: 'BiIntVector' and 'complex'
>>> v0 - v4
(0, 8)
>>> 0j - v4
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for -: 'complex' and 'BiIntVector'
>>>
```

下面的语句验证了BiIntVector对乘法和基于乘法的增强赋值运算的支持，注意虽然BiIntVector对象可以转化为复数，但不支持与复数相乘。此外，该乘法其实并非向量乘法，而是我们自定义的一种运算，它将向量的每个分量按对应位置相乘：

```
>>> v1 * v2
(9, -16)
>>> v5 = v2
>>> v5
(3, -4)
>>> v5 *= v1
augmented multiplication!
>>> v5
(9, -16)
>>> v5 * v0
(0, 0)
>>> v5 * 0j
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for *: 'BiIntVector' and 'complex'
>>>
```

下面的语句验证了BiIntVector对除法和基于除法的增强赋值运算的支持，注意虽然BiIntVector对象可以转化为复数，但不支持与复数相除。此外，该除法也是我们自定义的一种运算，它将向量的每个分量按对应位置进行整除：

```
>>> v1 / v2
(1, -1)
>>> v6 = v2
>>> v6
(3, -4)
>>> v6 /= v1
augmented division!
>>> v6
(1, -1)
>>> v6 / v0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Users/wwy/abc2.py", line 144, in __truediv__
```

```
          return BiIntVector(self.x // other.x, self.y // other.y)
                             ~~~~~~~^^~~~~~~~~
      ZeroDivisionError: integer division or modulo by zero
      >>> v6 / 0j
      Traceback (most recent call last):
        File "<stdin>", line 1, in <module>
      TypeError: unsupported operand type(s) for /: 'BiIntVector' and 'complex'
      >>> v0 / v6
      (0, 0)
      >>> 0j / v6
      Traceback (most recent call last):
        File "<stdin>", line 1, in <module>
      TypeError: unsupported operand type(s) for /: 'complex' and 'BiIntVector'
      >>>
```

下面的语句验证了BiIntVector对乘方和基于乘方的增强赋值运算的支持，注意虽然BiIntVector对象可以转化为复数，但不支持与复数乘方。此外，该乘方也是我们自定义的一种运算，它将向量的每个分量按对应位置进行乘方，然后再取整：

```
      >>> v1 ** v2
      (27, 0)
      >>> v7 = v2
      >>> v7
      (3, -4)
      >>> v7 **= v1
      augmented power!
      >>> v7
      (27, 256)
      >>> v7 ** v0
      (1, 1)
      >>> v7 ** 0j
      Traceback (most recent call last):
        File "<stdin>", line 1, in <module>
      TypeError: unsupported operand type(s) for ** or pow(): 'BiIntVector' and
'complex'
      >>> v0 ** v7
      (0, 0)
      >>> 0j ** v7
      Traceback (most recent call last):
        File "<stdin>", line 1, in <module>
      TypeError: unsupported operand type(s) for ** or pow(): 'complex' and
'BiIntVector'
      >>>
```

综上所述，BiIntVector虽然实现了Complex中的所有抽象方法和混入方法，但不能被看成是从复数派生的，这些方法也与复数的同名方法不完全兼容。所以我们通过register()将BiIntVector注册为Complex的虚子类，而不让它直接继承Complex是最合理的。

---

abc.ABCMeta重写的__subclasscheck__增加了第二个额外功能，使得可以通过在抽象基类中实现__subclasshook__魔术属性来控制其行为。__subclasshook__必须被实现为一个类方法，通过subclass参数传入被判断的类，而它会在__subclasscheck__中被自动调用，返回值可以是：

True：该类可以被视为该抽象基类的虚子类。

False：该类不能被视为该抽象基类的虚子类。
NotImplemented：继续后续的判断流程。
显然，__subclasscheck__被执行时__subclasshook__是最先被调用的，然后是检查通过register()注册的缓冲区，最后才是调用type实现的__subclasscheck__。值得一提的是，object实现了__subclasshook__，但它总是返回NotImplemented。

Python标准库中的collections.abc模块广泛使用了__subclasshook__。该模块与numbers类似，提供了用于实现Python中容器类型、可哈希类型、可调用类型、可等待类型和异步类型的抽象基类，其层次结构如图15-3所示。而表15-2则总结了所有这些抽象基类引入的抽象方法和混入方法。

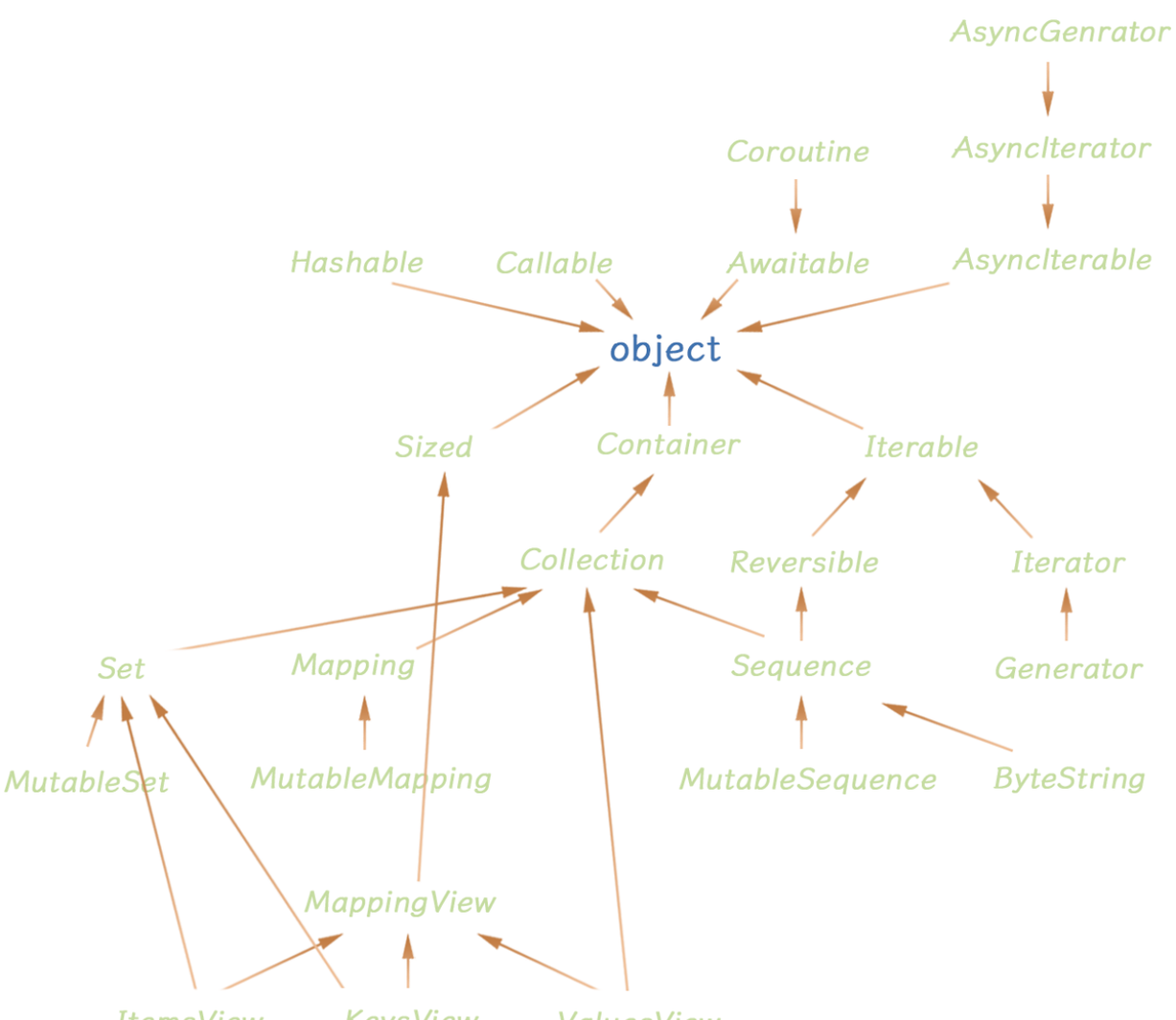

图15-3. collections.abc的抽象基类

表15-2. collections.abc中的抽象基类的接口

抽象基类	抽象方法	混入方法
Sized	__len__	无
Container	__contains__	无
Iterable	__iter__	无
Iterator	__next__	__iter__
Generator	send()、throw()	__next__、close()

抽象基类	抽象方法	混入方法
Reversible	__reversed__	无
Collection	无	无
Sequence	__getitem__	__contains__、__iter__、__reversed__、index()、count()
MutableSequence	__setitem__、__delitem__、insert()	__getitem__、__iadd__、append()、extend()、remove()、pop()、reverse()
ByteString	无	无
Mapping	__getitem__	__contains__、__eq__、__ne__、get()、keys()、items()、values()
MutableMapping	__setitem__、__delitem__	update()、setdefault()、pop()、popitem()、clear()
Set	无	__lt__、__le__、__eq__、__ne__、__gt__、__ge__、__and__、__xor__、__or__、__sub__、isdisjoint()
MutableSet	add()、discard()	__iand__、__ixor__、__ior__、__isub__、remove()、pop()、clear()
MappingView	无	__len__
ItemsView	无	__contains__、__iter__
KeysView	无	__contains__、__iter__
ValuesView	无	__contains__、__iter__
Hashable	__hash__	无
Callable	__call__	无
Awaitable	__await__	无
Coroutine	send()、throw()	close()
AsyncIterable	__aiter__	无
AsyncIterator	__anext__	__aiter__
AsyncGenerator	asend()、athrow()	aclose()、__anext__

collections.abc中的如下抽象基类重写了__subclasshook__：Sized、Container、Iterable、Iterator、Generator、Collection、Hashable、Callable、Awaitable、Coroutine、AsyncIterable、AsyncIterator、AsyncGenerator。通过这种方式，一个类只要实现了这些抽象基类规定的所有抽象方法和混入方法，就能自动被视为该抽象基类的虚子类，不需要通过register()注册。

需要注意的是，一个类可以被视为Iterable的虚子类，说明它实现了__iter__，因此肯定是可迭代的。但第12章已经说明，一个实现了__getitem__且索引从0开始计数的类也是可迭代的，不需要实现__iter__，因此存在不被视为Iterable的虚子类但可迭代的类。因此判断一个对象是否可迭代的最可靠方法是以它为参数调用内置函数iter()，并检查是否抛出了TypeError异常。

并非表15-2中所有抽象基类都重写了__subclasshook__，例如Sequence就没有。下面的例子自定义了一个继承Sequence的抽象基类MySequence，并通过重写__subclasshook__使得它能够自动将实现了Sequence要求的所有抽象方法和混入方法的类视为自己的虚子类，这会导致该类也被视为Sequence的虚子类：

```
import collections.abc


#定义继承Sequence的抽象基类。
class MySequence(collections.abc.Sequence):
    #重写__subclasshook__。
    @classmethod
    def __subclasshook__(cls, subclass):
        #仅当通过MySequence调用该属性时才需要做判断，通过MySequence的子类调用该
        # 属性时将直接返回NotImplemented。
        if cls is MySequence:
            #Sequence的虚子类必须是Collection的虚子类。
            if not issubclass(subclass, collections.abc.Collection):
                return False
            #Sequence的虚子类必须是Iterable的虚子类。
            if not issubclass(subclass, collections.abc.Iterable):
                return False
            #Sequence的虚子类还必须实现__reversed__、__getitem__、index()和
            # count()。
            if hasattr(subclass, "__reversed__") and\
                    hasattr(subclass, "__getitem__") and\
                    hasattr(subclass, "index") and\
                    hasattr(subclass, "count"):
                return True
        #无法通过上面的条件判断时，继续__subclasscheck__的后续步骤。
        return NotImplemented
```

请将上述代码保存为abc3.py。

为了验证自定义抽象基类MySequence的作用，下面自定义了两个容器类型：

```
import collections.abc


#自定义一个类，它基于映射实现不可变序列的基本功能，但没有实现__reversed__、index()和
# count()。
class SeqExample1(collections.abc.Mapping):
    #基于一个映射设置序列，键值对转换为序列对象的实例属性，并自动计算和存储序列长度。
    def __init__(self, d):
        self.length = 0
        for k, v in d.items():
            setattr(self, k, v)
            self.length += 1

    #基于实例属性动态生成一个列表，然后返回该列表的迭代器。
    def __iter__(self):
        l = []
        for key in self.__dict__:
            if key[0] != "_" and key != "length":
                l.append(getattr(self, key))
        return iter(l)

    #返回储存的序列长度。
```

```
        def __len__(self):
            return self.length

        #将键转化为实例属性。
        def __getitem__(self, key):
            return getattr(self, key)


    #自定义一个类，它继承了SeqExample1，额外实现了__reversed__、index()和count()。
    class SeqExample2(SeqExample1):
        #基于实例属性动态生成一个列表，将其反向，然后返回该列表的迭代器。
        def __reversed__(self):
            l = []
            for key in self.__dict__:
                if key[0] != "_" and key != "length":
                    l.append(getattr(self, key))
            l.reverse()
            return iter(l)

        #遍历实例属性以根据值查找键。
        def index(self, value):
            for key in self.__dict__:
                if key[0] != "_" and key != "length":
                    if getattr(self, key) == value:
                        return key
            raise ValueError(f"{value} not in sequence")

        #遍历实例属性以计算值的出现次数。
        def count(self, value):
            n = 0
            for key in self.__dict__:
                if key[0] != "_" and key != "length":
                    if getattr(self, key) == value:
                        n += 1
            return n
```

请将上述代码保存为abc4.py。

下面首先通过如下命令行和语句验证两种自定义容器类型的功能：

```
$ python3 -i abc4.py
>>> o1 = SeqExample1({"a": 1, "b": 2, "c": 3, "d": 2, "e": 3, "f": 2})
>>> o2 = SeqExample2({"a": 1, "b": 2, "c": 3, "d": 2, "e": 3, "f": 2})
>>> len(o1)
6
>>> len(o2)
6
>>> o1["a"]
1
>>> o2["b"]
2
>>> o1.c
3
>>> o2.d
2
>>> for n in o1:
...     print(n)
...
1
2
```

```
3
2
3
2
>>> for n in reversed(o2):
...     print(n)
...
2
3
2
3
2
1
>>> o2.index(2)
'b'
>>> o2.count(2)
3
>>> o2.index(4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Users/wwy/abc4.py", line 50, in index
    raise ValueError(f"{value} not in sequence")
    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
ValueError: 4 not in sequence
>>> o2.count(4)
0
>>>
```

然后请通过如下命令行和语句验证两种自定义容器类型在默认情况下都不被视为Sequence的虚子类：

```
$ python3 -i abc4.py

>>> issubclass(SeqExample1, collections.abc.Sequence)
False
>>> issubclass(SeqExample2, collections.abc.Sequence)
False
>>>
```

最后请通过如下命令行和语句验证，当定义了MySequence抽象基类后，SeqExample1仍然不被视为它和Sequence的虚子类，但SeqExmple2会同时被视为它和Sequence的虚子类：

```
$ python3 -i abc3.py

>>> from abc4 import *
>>> issubclass(SeqExample1, collections.abc.Sequence)
False
>>> issubclass(SeqExample2, collections.abc.Sequence)
True
>>> issubclass(SeqExample1, MySequence)
False
>>> issubclass(SeqExample2, MySequence)
True
>>>
```

上述结果说明，abc.ABCMeta的__subclasscheck__在执行__subclasshook__时，并不会直接调用指定抽象基类的__subclasshook__，而会非常智能地查找指定抽象基类有哪些派生抽象基类，然后依次调用这些派生抽象基类的__subclasshook__，最后才调用指定抽象基类自身的__subclasshook__。如果这一过程中有某个__subclasshook__返回了True或False，就不用再调用后续的__subclasshook__。

现在让我们思考这样一个问题：抽象基类的本质是具有抽象方法的类，然而类的属性是可以动态修改的。那么如果将抽象基类的所有抽象方法都替换为普通的方法、类方法和静态方法，该抽象基类会自动退化为类么？

为了回答这个问题，首先要介绍Python解释器是怎么判断一个类是否是抽象基类的。事实上，当我们通过@abc.abstractmethod将一个方法、类方法和静态方法装饰成抽象方法时，会自动为其添加__isabstractmethod__属性，并使该属性引用True。而该类定义语句被执行的过程中，只要Python解释器发现有一个方法具有该属性且引用True，就将该类视为抽象基类；否则，将该类视为普通的类。

下面通过abc1.py来证实__isabstractmethod__属性的存在：

```
$ python3 -i abc1.py

>>> getattr(D.d, "__isabstractmethod__", False)
True
>>> getattr(D.a_mixin, "__isabstractmethod__", False)
False
>>> getattr(F.d, "__isabstractmethod__", False)
False
>>> getattr(E.b, "__isabstractmethod__", False)
True
>>> getattr(E.c, "__isabstractmethod__", False)
True
>>> getattr(E.c_mixin, "__isabstractmethod__", False)
False
>>> getattr(G.b, "__isabstractmethod__", False)
False
>>> getattr(G.c, "__isabstractmethod__", False)
False
>>>
```

而当一个抽象基类的所有抽象方法都被替换后，该抽象基类并不会自动被识别为普通类，而需要调用abc模块提供的update_abstracemethods()，其语法为：

```
abc.update_abstractmethods(cls)
```

该函数会直接返回通过cls传入的类，因此可以被作为类装饰器使用。它的功能仅当该类的元类是abc.ABCMeta或其子类时才会显现——Python解释器会重新检查该类是否为抽象基类。

请继续上面的例子进行验证：

```
>>> D.d = F.d
>>> obj = D()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class D with abstract method d
>>> abc.update_abstractmethods(D)
<class '__main__.D'>
>>> obj = D()
>>> E.b = G.b
>>> E.c = G.c
>>> obj = E()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class E with abstract methods b, c
>>> abc.update_abstractmethods(E)
<class '__main__.E'>
>>> obj = E()
>>>
```

最后，请注意@abc.abstractmethod只能装饰方法、类方法和静态方法，当一个属性引用一个描述器，就不能通过@abc.abstractmethod来装饰。描述器本身也可以是一个抽象基类，在这种情况下该描述器自身必须提供引用True的__isabstractmethod__属性。

15-3. 标注与类型注解

（教程：4.8.8）
（语言参考手册：6.14、7.2.2、8.7）
（标准库：typing）

我们已经知道，Python同时是动态类型语言和强类型语言。这意味着我们在使用一个变量时不需要先声明它的类型，但在将其作为表达式的操作数或者函数调用的实际参数时却需要进行类型检查，并在类型不符合时抛出TypeError异常。由于Python真正做到的了一切皆对象，任何操作最终都会转化为对属性的访问，所以这种类型检查也是比较容易的实现的，即当该对象不具有所需的属性时，就说明其类型不对。这也是Python能同时为动态类型和强类型的原因，其他编程语言通常是静态类型与强类型搭配，动态类型与弱类型搭配。

然而对于Python编程者来说，能事先知道一个变量（包括函数的形式参数和返回值）的类型是很有必要的，尤其是当使用其他人开发的模块或扩展模块时。很长一段时间内，Python社区推荐将类型相关信息写在文档字符串或注释里，并开发了一些工具基于这些信息对脚本进行类型检查。当然这些类型检查都是静态的，不需要执行脚本。然而由于没有对这些信息的格式进行规范，这些工具的效果很难令人满意。这使得Python社区逐渐意识到，需要引入一种新的语法特性来解决这一问题。

被引入的语法特性就是“标注（annotation）”。它最初的设计目标不只包括支持类型检查，还包括外语支持、数据库查询映射、RPC参数编组等等。然而目前看来在实践中标注几乎只被用于支持类型检查。当标注中包含的信息是关于类型之时，这些信息又被称为“类型注解（type hints）”。

最初标注只是针对函数的，用于说明其形式参数和返回值的类型，被称为“函数标注（function annotations）”，通过PEP 3107规范。下面是一个包含标注的函数的例子：

```
#定义一个重复字符串的函数，并通过标注说明：
#   形式参数s需被传入一个字符串。
#   形式参数n需被传入一个整数。
#   返回值是一个字符串。
def str_multiplier(s: str, n: int = 2) -> str:
    return s * n

#调用该函数，并按标注的要求传入实际参数。
print(str_multiplier("abc"))

#调用该函数，给s传入一个二进制串。
print(str_multiplier(b"01", 3))

#调用该函数，给s传入一个整数，给n传入一个浮点数。
print(str_multiplier(10, 4.5))
```

请将上述代码保存为annotation1.py，然后通过如下命令行和语句验证：

```
$ python3 -i annotation1.py
abcabc
b'010101'
45.0
>>> str_multiplier.__annotations__
{'s': <class 'str'>, 'n': <class 'int'>, 'return': <class 'str'>}
>>>
```

现在让我们对上面的例子进行分析。首先，str_multiplier()的定义语句说明了函数标注的语法规则：

➤ 形式参数的变量名之后跟一个冒号，然后可以跟任何表达式。但仅当该表达式的求值结果代表一个类型（会在下一节详细解释）时，才能被视为类型注解。
➤ 如果形式参数具有默认实参值，则应写在标注之后。
➤ 函数的返回值的标注在函数的形式参数列表后面（“)”之后）通过“->”给出，也是一个表达式，与形式参数的标注要求相同。而函数定义语句原本就有的冒号应放到该表达式之后。

此外，当使用了任意实参列表技术时，args和kwargs也可以被当成一个形式参数，给出一个标注，只不过当该标注为类型注解时，将限制所有与该形式参数匹配的实际参数。

上面的例子也给出了PEP 8推荐的标注格式，即：

- “:”前面没有空格，后面跟一个空格。
- 如果存在默认实参值，则“=”的两边各有一个空格。
- “->”前后都有一个空格。
- 返回值的标注后面紧跟“:”。

此外，函数标注不要求给每个形式参数都添加标注，但一般而言，建议函数标注包含所有形式参数的标注，以及返回值的标注（仅当返回值是None时可以省略）。

而上面例子的结果说明，即使添加了类型注解，在运行时给函数传入不符合其指定类型的实际参数，返回值也不符合其指定类型时，依然不会抛出任何异常。事实上，标注与文档字符串类似，是不会影响脚本的执行的，而只是为专门的工具提供所需信息，对于类型注解来说就是类型检查工具。

最后，该结果还说明，函数标注的所有信息都会被储存到该函数的__annotations__特殊属性中。__annotations__引用一个字典，该字典会在函数定义语句被执行的过程中被填充，即为每个具有标注的形式参数创建一个键值对，键即形式参数的变量名，值即对标注表达式求值得到的结果；如果返回值也具有标注，则同样增加一个键值对，键是“return”，值依然是对标注表达式求值得到的结果。注意“return”不可能被作为形式参数，否则会抛出SyntaxError异常。

我们已经知道类型注解仅在使用类型检查工具时才有意义。常用的静态类型检查工具包括mypy（Python官方提供）、pytype（Google提供）、pyre（Facebook提供）和pyright（Microsoft提供）等等，而使用Python集成开发环境PyCharm时也会自动进行静态类型检查。

下面以mypy为例子说明如何进行静态类型检查。首先通过如下命令行安装mypy：

```
$ python3 -m pip install --upgrade pip setuptools
$ python3 -m pip install --upgrade mypy
```

然后通过如下命令行进行类型检查：

```
$ mypy annotation1.py
annotation1.py:12: error: Argument 1 to "str_multiplier" has incompatible type "bytes"; expected "str"  [arg-type]
annotation1.py:15: error: Argument 1 to "str_multiplier" has incompatible type "int"; expected "str"  [arg-type]
annotation1.py:15: error: Argument 2 to "str_multiplier" has incompatible type "float"; expected "int"  [arg-type]
Found 3 errors in 1 file (checked 1 source file)
```

该结果说明，mypy找到了3处违反类型注解的参数传递。

关于函数标注还有一点需要说明的是，匿名函数的形式参数和返回值都不能具有标注。

PEP 526将函数标注的理念扩展到了任意变量，使得标注可以出现在赋值语句中（但只能位于“=”的左边），还可以把具有标注的变量单独写为一行（类似于C中的变量声明）。此类标注被称为“变量标注（variable annotations）”。

下面的例子说明了如果给全局变量、本地变量、类变量（类的非函数属性）和实例变量（实例的非函数属性）添加变量标注：

```
import random
import typing

#该全局变量的标注说明它应该引用一个浮点数。
m: float = 1.0


#定义一个函数，该函数本身没有标注，但其内的本地变量具有标注。
def arith_seq(n):
    #声明m是全局变量。这里不能有标注。
    global m

    #该本地变量的标注说明它应该引用一个整数。
    start: int = random.randrange(3)

    #该本地变量的标注说明它应该引用一个整数列表。
    seq: list[int] = []

    #for语句中的变量不能有标注，故只能将标注单独写一行。    i的标注说明它应该引用一个
    # 复数，但被赋值整数，不符合其标注。
    i: complex
    for i in range(n):
        #seq被添加浮点数，不符合其标注。
        seq.append(start + i*m)

    def result():
        #声明seq是外层本地变量。这里不能有标注。
        nonlocal seq
        print(seq)

    return result


#m的标注被修改，但这在类型检查中会被视为错误，不起作用。
m: complex
#m被赋值一个复数，不符合其标注。
m = 2 + 0j


#定义一个类。
class GeoSeq():
    #该类变量的标注说明它应该引用一个浮点数，且被所有GeoSeq类的对象共享。
    #   它被赋值一个复数，不符合其标注。
    m: typing.ClassVar[float] = m
```

```
        #该类变量的标注说明它应该引用一个整数，且被当成同名实例变量的默认值使用。
        start: int = 1

        #__init__方法中的本地变量就是实例变量。
        def __init__(self, n):
            #该实例变量的标注说明它应该引用一个整数。
            self.n: int = n
            #该实例变量的标注说明它应该引用一个整数。
            self.start: int = random.randrange(3)
            if self.start == 0:
                self.start = GeoSeq.start

        #该方法中的本地变量也有标注。
        def __enter__(self):
            #该本地变量的标注说明它应该引用一个浮点数列表。
            seq: list[float] = []
            #同样，for语句中的变量不能有标注。   i的标注说明它应该引用一个整数。
            i: int
            for i in range(self.n):
               seq.append(self.start * GeoSeq.m**i)
            return seq

        def __exit__(self, exc_type, exc_value, traceback):
            return False


    #显示等差数列。
    f = arith_seq(5)
    f()

    #显示等比数列。
    o = GeoSeq(5)
    #with语句中的变量不能有标注，故只能将标注单独写一行。   result的标注说明它应该引用一个
    # 字符串列表，但实际被赋值一个复数列表，不符合其标注。
    result: list[str]
    with o as result:
        #解包赋值中的变量不能有标注，故只能分别将每个变量的标注单独写一行。   这些变量标注
        # 说明它们都应引用一个浮点数，但实际都被赋值一个复数，不符合其标注。
        a1: float
        a2: float
        a3: float
        a4: float
        a5: float
        a1, a2, a3, a4, a5 = result
        print(f"[{a1}, {a2}, {a3}, {a4}, {a5}]")
```

请将上述代码保存为annotation2.py，然后通过如下命令行和语句验证：

```
    $ python3 -i annotation2.py
    [(1+0j), (3+0j), (5+0j), (7+0j), (9+0j)]
    [(1+0j), (2+0j), (4+0j), (8+0j), (16+0j)]
    >>> import __main__
    >>> __main__.__annotations__
    {{'m': <class 'complex'>, 'result': list[str], 'a1': <class 'float'>, 'a2':
<class 'float'>, 'a3': <class 'float'>, 'a4': <class 'float'>, 'a5': <class
'float'>}
    >>> arith_seq.__annotations__
    {}
    >>> GeoSeq.__annotations__
    {'m': typing.ClassVar[float], 'start': <class 'int'>}
```

```
>>> GeoSeq.__init__.__annotations__
{}
>>> GeoSeq.__enter__.__annotations__
{}
>>>
```

现在让我们对上面的例子进行分析。“m: float = 1.0”和“result: list[str]”说明了如何给全局变量添加标注。arith_seq()中关于start和seq的赋值语句，以及“i: complex”说明了如何给函数中的本地变量添加标注。GeoSeq中关于m和start的赋值语句说明了如何给类变量添加标注。GeoSeq.__init__中关于self.n和self.start的赋值语句说明了如何给实例变量添加标注。GeoSeq.__enter__关于seq的赋值语句和“i: int”说明了如何给方法中的本地变量添加标注。可以看出，所有这些变量标注的语法都与函数标注中形式参数的标注相同。

全局变量的变量标注会被求值，并被保存到相应模块的__annotations__特殊属性引用的字典中。类变量的变量标注也会被求值，并被保存到该类的__annotations__特殊属性引用的字典中。而不论是函数中的本地变量还是方法中的本地变量（包括本质为__init__中的本地变量的实例变量），其标注都不会被求值，也不会被保存到相应函数的__annotations__特殊属性引用的字典中，因为该字典是用来保存函数标注的。然而虽然本地变量的标注不会被求值，但会强制使Python解释器将其视为本地变量，哪怕它没有被赋值就使用（这会导致抛出UnboundLocalError异常）。

在用global关键字声明全局变量时（例如arith_seq()中的“global m”），或用nonlocal关键字声明非本地变量时（例如arith_seq()中的“nonlocal seq”），不能给它们添加变量标注。这是因为这些变量其实是在其余位置定义的，而我们只能在一个变量被定义处给它添加标注。此外，如果在多处给同一个变量提供了标注（例如上面例子中的“m: float = 1.0”和“m: complex”），则对于Python解释器来说最后一次标注有效，但在进行类型检查时通常会报错。

for语句中的变量不能具有标注，因此只能在for语句之前给出该变量的标注，并将其单独写成一行。with语句中的变量也不能具有标注，因此只能在with语句之前给出该变量的标注，并将其单独写成一行。同理，解包赋值中的变量也不能具有标注，必须将其中每个变量的标注单独写成一行。

最后值得说明的是，类变量Geo.m的标注使用了标准库中的typing模块提供的ClassVar结构，该结构的语法为：

```
typing.ClassVar[Type]
```

其中Type必须是一个不包含类型变量（会在下节解释）的类型表达式。typing.ClassVar的含义是，该类变量应该具有Type类型，且该类的实例不能有同名的实例变量，以避免该类变量被屏蔽。这说明该类变量储存的是运行时被该类的所有实例共享的数据，而并非只是一个默认值，因此是一个“真正的类变量”。而类变量Geo.start由于只是被当成默认值使用，所以其标注中没有typing.ClassVar。

接下来用mypy对annotation2.py进行类型检查：

```
      $ mypy annotation2.py
      annotation2.py:14: note: By default the bodies of untyped functions are not
checked, consider using --check-untyped-defs  [annotation-unchecked]
      annotation2.py:17: note: By default the bodies of untyped functions are not
checked, consider using --check-untyped-defs  [annotation-unchecked]
      annotation2.py:21: note: By default the bodies of untyped functions are not
checked, consider using --check-untyped-defs  [annotation-unchecked]
      annotation2.py:35: error: Name "m" already defined on line 5  [no-redef]
      annotation2.py:37: error: Incompatible types in assignment (expression has
type "complex", variable has type "float")  [assignment]
      annotation2.py:52: note: By default the bodies of untyped functions are not
checked, consider using --check-untyped-defs  [annotation-unchecked]
      annotation2.py:54: note: By default the bodies of untyped functions are not
checked, consider using --check-untyped-defs  [annotation-unchecked]
      annotation2.py:61: note: By default the bodies of untyped functions are not
checked, consider using --check-untyped-defs  [annotation-unchecked]
      annotation2.py:63: note: By default the bodies of untyped functions are not
checked, consider using --check-untyped-defs  [annotation-unchecked]
      annotation2.py:89: error: Incompatible types in assignment (expression has
type "str", variable has type "float")  [assignment]
      Found 3 errors in 1 file (checked 1 source file)
```

注意annotation2.py中类型不符合标注的地方其实有6处，而mypy只找到其中三处，且第三处对result中元素的类型的判断还是错的。这是因为mypy只能进行静态类型检查，对于需要执行脚本才能确定类型的情况无能为力。相对于静态类型检查，实现动态类型检查的难度要高几个数量级，因此目前还不存在像mypy这样简单易用的针对Python的动态类型检查工具，我们需要自己编写进行动态类型检查的代码并将它们嵌入脚本。

不能被静态类型检查利用的标注并非没有用处。PyCharm会基于这些标注给出提示。很多第三方工具会在运行时利用这些标注实现其核心功能，例如编写框架时可以使用pydantic和valideer之类的数据接口验证模块，它们能够基于数据接口的标注实现局部的动态类型检查，并在发现不符合标注的情况时抛出异常。

由于静态类型检查工具和动态利用标注的工具对同一函数或变量的标注的内容很可能有不同的要求，所以PEP 593为typing模块增添了Annotated结构，其语法为：

```
    typing.Annotated[Type, MetaData, ...]
```

其中Type必须是一个类型表达式，后面则可以跟任意多个表达式，它们的求值结果可以为任何对象，提供的信息被视为解释Type的元数据。下面是一个例子：

```
    T1 = typing.Annotated[int, ValueRange(-30, 50)]
```

容易看出，T1代表着-30～50范围内的整数。当利用标注的工具遇到Annotated结构时，可以忽略不能识别的元数据，仅使用类型表达式和能够识别的元数据。

Annotated结构具有如下性质：
1. 允许嵌套，但嵌套会被自动展平，例如：

```
typing.Annotated[typing.Annotated[int, ValueRange(-30, 50)], ctyp("char")]
```

就等价于

```
typing.Annotated[int, ValueRange(-30, 50), ctyp("char")]
```

2. 从1可以推理出，允许以之前定义过的Annotated结构作为类型表达式，等同于嵌套，例如：

```
T2 = typing.Annotated[T1, ctype("char")]
```

等同于

```
T2 = typing.Annotated[typing.Annotated[int, ValueRange(-30, 50)],
ctyp("char")]
```

并进一步等同于

```
T2 = typing.Annotated[int, ValueRange(-30, 50), ctyp("char")]
```

3. 元数据的顺序是有影响的，例如：

```
typing.Annotated[int, ValueRange(-30, 50), ctype("char")]
```

并不等同于

```
typing.Annotated[int, ctype("char"), ValueRange(-30, 50)]
```

4. 从2可以推理出，重复的元数据不能被合并，例如：

```
typing.Annotated[int, ctype("char"), ctype("char"), ValueRange(-30, 50)]
```

并不等同于

```
typing.Annotated[int, ctype("char"), ValueRange(-30, 50)]
```

至此已经两次提到了标准库中的typing模块，该模块专门用于提供对Python类型的支持，对于类型注解，以及后面将要讨论的泛型和多态来说具有至关重要的作用。本节和后续五节会详细讨论typing模块的功能。

typing提供了一个常量typing.TYPE_CHECKING，Python解释器会将其解读为False，而静态类型检查工具则会将其解读为True。这使得我们可以通过如下方式在脚本中嵌入仅在静态类型检查时被执行的代码：

```
if typing.TYPE_CHECKING:
    ... any code goes here ...
```

我们已经知道，标注本质上是一个表达式。当该标注是类型注解时，如果它引用了前面定义的标识符，且它会被求值（即它不是本地变量的标注）时，必须将该标识符用引号括起来，并被视为一个“前向引用（forward reference）”。举例来说，下面的类型注解说明seq_set应该引用一个由GeoSeq类的实例形成的凝固集合：

```
seq_set: frozenset["GeoSeq"]
```

其中的“GeoSeq”就是一个前向引用。

typing提供了ForwardRef类来支持前向引用。我们永远不需要手工实例化该类，因为它是在解析前向引用时被自动使用的。对于上面的例子来说，frozenset["GeoSeq"]会被解析成frozenset[ForwardRef("GeoSeq")]。而ForwardRef对象最终会被解析为相应标识符引用的对象。

typing提供了get_type_hints()函数，使我们不需要直接访问__annotations__特殊属性来了解一个模块、类或函数的标注。该函数的语法为：

```
typing.get_type_hints(obj, globals=None, locals=None, include_extras=False)
```

其中obj参数用于指定模块、类或函数；globals参数和locals参数的含义与在eval()和exec()中相同，仅在解析前向引用时有意义；include_extras参数仅在解析Annotated结构时有意义。该函数的返回值与__annotations__引用的字典存在如下区别：

➤ 如果obj参数被传入的是类，则会合并它的__annotations__引用的字典与它的所有基类的__annotations__引用的字典。
➤ 如果标注中包含前向引用，则会将其解析为相应标识符引用的对象，该过程中可以通过globals参数指定全局名字空间，locals参数指定本地名字空间。
➤ 如果include_extras参数被传入True，则Annotated结构会被保留；否则，Annotated结构会被替换成其第一项类型表达式的求值结果。

typing还提供了如下三个装饰器来影响类型检查工具的行为：

➤ @typing.no_type_check：装饰函数或类，表明函数或类中的标注都不是类型提示。
➤ @typing.type_check_only：装饰函数或类，表明函数或类中的标注只能用于静态类型检查，不能在运行时被使用。
➤ @typing.no_type_check_decorator：装饰另一个装饰器，使其具有no_type_check的效果。

当@typing.no_type_check和@typing.type_check_only被当成类装饰器来使用时，将递归地作用于该类的所有方法，以及在该类的类体中定义的其他类，然而不会影响到该类的基类和子类。

接下来说明什么是“存根文件（stub files）”。存根文件是PEP 484引入的一个概念，最重要的用处是为扩展模块提供记录标注的机制（因为CPython提供的C API无法实现标注）。当然，模块也可以具有存根文件。存根文件的文件名必须是模块名，而后缀名推荐用.pyi，以使得它们可以和实现模块的文件放在同一目录下。如果不将存根文件与实现模块的文件放在同一目录下，那么需要配置静态类型检查工具使得它们能够在存放存根文件的目录下搜索。为了使用存根文件，或者通过-m选项指定模块名，或者直接给出存根文件的文件名。

存根文件只能被静态类型检查工具使用，其内容为相关模块中带类型注解的类定义、函数定义和全局变量定义。存根文件使用标准Python语法，但建议用省略号替代函数体（因为静态类型检查工具不需要查看它们），对全局变量的定义也可以通过标注本身实现（不需要写成赋值语句）。

下面的例子改编自Pocker.py，说明了如何使用上述常量、函数和装饰器：

```
    import functools
    import typing
    import random


    #该类的实例代表扑克牌。
    @functools.total_ordering
    class Pocker:
        #下列类变量的标注说明它们都应该引用一个字符串。
        SPADE: str = "\u2664"
        HEART: str = "\u2661"
        CLUB: str = "\u2667"
        DIAMOND: str = "\u2662"

        #suit参数的标注说明它应该被传入一个1~4范围内的整数。   rank参数的标注说明它应该被传
        # 入一个1~13范围内的整数。   该方法创建了一张扑克牌。
        def __init__(self, suit: typing.Annotated[int, range(1, 5)], rank:
typing.Annotated[int, range(1, 14)]):
            self.suit = int(suit)
            if self.suit > 4:
                self.suit = 4
            if self.suit < 1:
                self.suit = 1
            self.rank = int(rank)
            if self.rank > 13:
                self.rank = 13
            if self.rank < 1:
                self.rank = 1

        #该方法和下面的__str__用于显示扑克牌。
        def __repr__(self):
            if self.suit == 1:
                s = Pocker.SPADE
            elif self.suit == 2:
                s = Pocker.HEART
            elif self.suit == 3:
                s = Pocker.CLUB
            else:
                s = Pocker.DIAMOND
            if self.rank == 1:
                s = s + ' A'
            elif self.rank == 11:
                s = s + ' J'
            elif self.rank == 12:
                s = s + ' Q'
            elif self.rank == 13:
                s = s + ' K'
            else:
                s = s + ' ' + str(self.rank)
            return s

        __str__ = __repr__

        #other参数的标注说明它应该被传入另一个Pocker对象。   该方法支持扑克牌进行相等比较。
        def __eq__(self, other: "Pocker"):
            if self.suit == other.suit and self.rank == other.rank:
                return True
            else:
                return False
```

```
        #other参数的标注说明它应该被传入另一个Pocker对象。    该方法支持扑克牌进行大小比较。
        def __lt__(self, other: "Pocker"):
            srank = self.rank
            if srank == 1:
                srank = 14
            orank = other.rank
            if orank == 1:
                orank = 14
            if srank < orank:
                return True
            elif srank > orank:
                return False
            else:
                if self.suit <= other.suit:
                    return False
                else:
                    return True


    #该类的实例代表一次抽牌的动作。
    class Draw(Pocker):
        #who参数的标注说明它应该被传入一个字符串。    该方法记录下来了谁（通过who传入的人名）
        # 抽中了哪张扑克牌。
        def __init__(self, who: str):
            self.who = who
            self.card = Pocker(random.randrange(1, 5), random.randrange(1, 14))

        #other参数的标注说明它应该被传入另一个Draw对象，返回值是整数。    该方法比较两次抽牌
        # 的结果，并判断谁获胜。
        def compare(self, other: "Draw") -> int:
            print(f"{self.who}'s card: {self.card}")
            print(f"{other.who}'s card: {other.card}")
            if self.card > other.card:
                print(f"{self.who} win!")
                return 1
            elif self.card < other.card:
                print(f"{other.who} win!")
                return -1
            else:
                print("A draw!")
                return 0


    #仅当进行静态类型检测时执行。
    if typing.TYPE_CHECKING:
        #三次抽牌。
        draw1 = Draw("Jimmy")
        draw2 = Draw("Nacy")
        #这次抽牌传入的人名是None，不符合标注。
        draw3 = Draw(None)
        #比较三次抽牌的结构。
        draw1.compare(draw2)
        print("")
        draw2.compare(draw3)
        print("")
        draw3.compare(draw1)
```

请将上述代码保存为annotation3.py，然后通过如下命令行和语句验证它可以正常执行：

```
    $ python3 -i annotation3.py
```

```
>>> draw1 = Draw("Lucy")
>>> draw2 = Draw("Lily")
>>> draw1.compare(draw2)
Lucy's card: ♤ 2
Lily's card: ♢ A
Lily win!
-1
>>>
>>> draw1 = Draw("Lucy")
>>> draw2 = Draw("Lily")
>>> draw1.compare(draw2)
Lucy's card: ♧ J
Lily's card: ♡ 9
Lucy win!
1
>>>
```

注意此时“if typing.TYPE_CHECKING”下的代码块并未被执行。

接下来验证调用typing.get_type_hints()与访问__annotations__的区别。下面的结果说明一般而言两者的结果是相同的：

```
>>> Pocker.__annotations__
{'SPADE': <class 'str'>, 'HEART': <class 'str'>, 'CLUB': <class 'str'>,
'DIAMOND': <class 'str'>}
>>> typing.get_type_hints(Pocker)
{'SPADE': <class 'str'>, 'HEART': <class 'str'>, 'CLUB': <class 'str'>,
'DIAMOND': <class 'str'>}
>>>
>>> Draw.__init__.__annotations__
{'who': <class 'str'>}
>>> typing.get_type_hints(Draw.__init__)
{'who': <class 'str'>}
>>>
```

下面的结果说明typing.get_type_hints()会合并一个类的标注和它的所有基类的标注：

```
>>> Draw.__annotations__
{}
>>> typing.get_type_hints(Draw)
{'SPADE': <class 'str'>, 'HEART': <class 'str'>, 'CLUB': <class 'str'>,
'DIAMOND': <class 'str'>}
>>>
```

下面的结果说明typing.get_type_hints()会解析前向引用：

```
>>> Pocker.__eq__.__annotations__
{'other': 'Pocker'}
>>> typing.get_type_hints(Pocker.__eq__)
{'other': <class '__main__.Pocker'>}
>>>
>>> Pocker.__lt__.__annotations__
{'other': 'Pocker'}
```

```
>>> typing.get_type_hints(Pocker.__lt__)
{'other': <class '__main__.Pocker'>}
>>>
>>> Draw.compare.__annotations__
{'other': 'Draw', 'return': <class 'int'>}
>>> typing.get_type_hints(Draw.compare)
{'other': <class '__main__.Draw'>, 'return': <class 'int'>}
>>>
```

下面的结果说明typing.get_type_hints()默认会将Annotated结构替换成类型，仅当给其include_extras参数传入True时才会保留Annotated结构：

```
>>> Pocker.__init__.__annotations__
{'suit': typing.Annotated[int, range(1, 5)], 'rank': typing.Annotated[int,
range(1, 14)]}
>>> typing.get_type_hints(Pocker.__init__)
{'suit': <class 'int'>, 'rank': <class 'int'>}
>>> typing.get_type_hints(Pocker.__init__, include_extras=True)
{'suit': typing.Annotated[int, range(1, 5)], 'rank': typing.Annotated[int,
range(1, 14)]}
>>>
```

接下来用mypy对模块annotation3进行静态类型检查：

```
$ mypy -m annotation3
annotation3.py:54: error: Argument 1 of "__eq__" is incompatible with
supertype "object"; supertype defines the argument type as "object"  [override]
annotation3.py:54: note: This violates the Liskov substitution principle
annotation3.py:54: note: See https://mypy.readthedocs.io/en/stable/
common_issues.html#incompatible-overrides
annotation3.py:54: note: It is recommended for "__eq__" to work with
arbitrary objects, for example:
annotation3.py:54: note:     def __eq__(self, other: object) -> bool:
annotation3.py:54: note:         if not isinstance(other, Pocker):
annotation3.py:54: note:             return NotImplemented
annotation3.py:54: note:         return <logic to compare two Pocker
instances>
annotation3.py:109: error: Argument 1 to "Draw" has incompatible type
"None"; expected "str"  [arg-type]
Found 2 errors in 1 file (checked 1 source file)
```

该结果说明mypy找到了两处类型错误。第一处是Pocker.__eq__的other参数被指定为传入另一个Pocker对象，使得它只能比较两个Pocker对象；而object中的__eq__是可以比较任意两个对象的。这个错误可以这样改正：先判断other的类型，当它不是Pocker对象时通过super()调用object的__eq__。但在本例子中这是有意为之，并不应被视为类型错误。而第二处是“draw3 = Draw(None)”，它给Draw.__init__传入了None，这是一个我们期望被发现的类型错误。注意这说明此时“if typing.TYPE_CHECKING”下的代码块被执行。

下面让我们为annotation3模块编写存根文件：

```
import functools
import typing
import random

@functools.total_ordering
class Pocker:
    SPADE: str
    HEART: str
    CLUB: str
    DIAMOND: str

    def __init__(self, suit: typing.Annotated[int, range(1, 5)], rank:
typing.Annotated[int, range(1, 14)]): ...

    def __repr__(self):...

    __str__ = __repr__

    #使用装饰器@typing.no_type_check以跳过静态类型检查。
    @typing.no_type_check
    def __eq__(self, other: "Pocker"): ...

    def __lt__(self, other: "Pocker"): ...


class Draw(Pocker):
    def __init__(self, who: str): ...

    def compare(self, other: "Draw") -> int: ...


if typing.TYPE_CHECKING:
    draw1 = Draw("Jimmy")
    draw2 = Draw("Nacy")
    draw3 = Draw(None)
    draw1.compare(draw2)
    print("")
    draw2.compare(draw3)
    print("")
    draw3.compare(draw1)
```

注意在存根文件中，所有方法的函数体都被替换成了“...”，极大地减少了代码量。

请将上述代码保存为annotation3.pyi，然后将其和annotation3.py放在同一个目录下。再次用mypy对模块annotation3进行静态类型检查：

```
$ mypy -m annotation3
annotation3.pyi:34: error: Argument 1 to "Draw" has incompatible type
"None"; expected "str"  [arg-type]
Found 1 error in 1 file (checked 1 source file)
```

该结果说明，当存在存根文件时，mypy会优先使用存根文件；而@typing.no_type_check装饰器成功使mypy忽略了Pocker.__eq__中的标注。

不论是脚本还是存根文件，当通过import语句将其他模块导入时，仅会对被导入的标识符进行类型检查。

最后需要强调，虽然标注和类型注解在维护大型项目时是比较有用的，但重要性是低于文档字符串的。我们完全可以在文档字符串中包含任何能写在标注中的信息，而相对于类型检查工具和其他利用标注的工具，基于文档字符串生成的帮组文档会被更多用户看到。基于这个原因，PEP 8建议我们在代码中广泛使用文档字符串，但没有建议我们在代码中广泛使用标注和类型注解。

15-4. 类型理论——基本类型的表示

（标准库：typing）

我们已经知道，标注最成功的应用是类型注解。这使得Python官方引入一套描述类型的方法，即Python的类型理论。这套理论以PEP 483、PEP 484和PEP 544为核心，后来又经多个PEP扩展，直到现在还在不断丰富其特性（Python 3.11引入了相当多关于类型理论的新特性）。本章将用5节的篇幅详细阐述类型理论。本节讨论如何表示基本的类型。

首先，任何内置类型的构造函数的函数名都可用于表示该内置类型，包括：bool、int、float、complex、str、bytes、bytearray、memoryview、tuple、list、range、dict、set和frozenset。用它们作为类型注解时在任何情况下都不需要用引号括起来。标准解释器类型没有构造函数，因此不能通过这种方式表示。但只有一个值的None、NotImplemented和Ellipsis可以直接用值来表示。

其次，用户自定义类，以及标准库中定义的类，都可以通过类名表示。大多数情况下，我们都可以直接使用这些类的类名。唯一需要注意的特殊情况，是在类体中的类型注解包含该类体所属类本身的类名，由于在类体被执行时该类尚未被创建，所以必须用引号将该类名括起来，以表明这是一个前向引用。annotation3.py中Pocker.__eq__、Pocker.__lt__和Draw.compare()中的类型注解就属于这种情况。

值得一提的是，collections模块提供了工厂函数namedtuple()来创建具名元组，导致具名元组没有相应类名。为了表示具名元组，typing模块引入了NamedTuple类。这也使得我们可以通过实例化该类或该类派生的子类来创建具名元组。由于本书不详细讨论collections模块，所以对于typing.NamedTuple也不详细讨论。

Python中的容器类型除了字符串、字节串、字节数组和内存视图外，其元素都可以具有不相同的类型。这就使得我们无法简单的用list、dict、set等类型名来精确表示此类容器类型。事实上，目前还没有办法精确表示所有属于这类情况的类型。但我们可以利用typing模块提供的Tuple结构和TypedDict()函数表示其中很少一部分。

Tuple结构的语法为：

```
typing.Tuple[type1, type2, ..., typeN]
typing.Tuple[type, ...]
typing.Tuple[()]
```

第一种语法表示一个具有N个元素的元组，其第一个元素的类型是type1，第二个元素的类型是type2，……，第N个元素的类型是typeN。第二种语法表示元素个数不限，元素类型都为type的元组。第三种语法表示空元组。

下面通过一个例子来验证Tuple结构的用法：

```
import typing

#该全局变量的标注说明它应该引用Tuple[int, str]类型的对象。
a: typing.Tuple[int, str]

#该赋值语句不会报错，因为符合标注。
a = (0, 'a')
#该赋值语句会报错，因为值的类型为List[object]。
a = [0, 'a']
#该赋值语句会报错，因为值的类型为Tuple[int]。
a = (0,)
#该赋值语句会报错，因为值的类型为Tuple[int, str, None]。
a = (0, 'a', None)
#该赋值语句会报错，因为值的类型为Tuple[float, bytes]。
a = (0.0, b'a')
```

请将上述代码保存为typing1.pyi，然后通过如下命令行验证：

```
$ mypy typing1.pyi
typing1.pyi:9: error: Incompatible types in assignment (expression has type
"List[object]", variable has type "Tuple[int, str]")  [assignment]
typing1.pyi:11: error: Incompatible types in assignment (expression has
type "Tuple[int]", variable has type "Tuple[int, str]")  [assignment]
typing1.pyi:13: error: Incompatible types in assignment (expression has
type "Tuple[int, str, None]", variable has type "Tuple[int, str]")  [assignment]
typing1.pyi:15: error: Incompatible types in assignment (expression has
type "Tuple[float, bytes]", variable has type "Tuple[int, str]")  [assignment]
Found 4 errors in 1 file (checked 1 source file)
```

上面的结果中还出现了“List[object]”，它其实就等价于list。因为object是任意对象所属类型的基类，所以它可以匹配元素是任意内置类型和类的列表。但需要强调的是，object并不能匹配任意类型，例如Tuple[int, str]就不能被它匹配。类型理论中的类型的范围超过了内置类型和类，这是需要特别强调的。

TypedDict()的语法为：

```
typing.TypedDict(clsname, dict, total=True)
```

其中clsname参数需被传入一个字符串；dict参数需被传入一个类型字典，即以字符串为键，以某个类型为值的字典；total参数需被传入一个布尔值。typing.TypedDict本质上是一种类型构造器，它的返回值是dict的一个直接子类，类名为clsname。dict参数指定的字典用于给该类的类属性添加标注：每个键值对的键代表一个类属性，值则代表该类属性的类型注解。当然由于该类作为dict的子类依然代表一个映射，所以它的类属性也可以被理解为键值对。当total参数被传入True时，dict参数指定的类属性都是必须的；当其被传入False时，dict参数指定的类属性都是可选的。不论哪种情况，该类都不能具有未被dict参数指定的类属性。（当然，这里的“必须”和“可选”都是相对于静态类型检查而言，即便没有这些类属性也不影响脚本本身的执行。）

下面请通过如下命令行和语句来验证typing.TypedDict的基本用法：

```
      $ python3

      >>> import time, typing
      >>> Message = typing.TypedDict("Message", {"uid": int, "time":
time.struct_time, "content": str}, total=False)
      >>> typing.is_typeddict(Message)
      True
      >>> type(Message)
      <class 'typing._TypedDictMeta'>
      >>> Message.__mro__
      (<class '__main__.Message'>, <class 'dict'>, <class 'object'>)
      >>> typing.get_type_hints(Message)
      {'uid': <class 'int'>, 'time': <class 'time.struct_time'>, 'content':
<class 'str'>}
      >>> msg = Message(uid=0, content="Hello")
      >>> msg
      {'uid': 0, 'content': 'Hello'}
      >>>
```

该结果说明，typing.TypedDict()返回的类的元类是typing._TypedDictMeta，而由于该类是dict的子类，所以可以通过创建字典的语法来实例化该类。

此外，上面的例子中还用到了typing.is_typeddict()函数，该函数的功能是检查一个类型是否是通过typing.TypedDict()创建的，语法为：

```
typing.is_typeddict(type)
```

其中type参数需传入被检查的类型。该函数返回True说明type是通过typing.TypedDict()创建的，否则返回False。

特别的，虽然typing.TypedDict()是一个函数，但我们可以将其当成一个基类来使用，也就是说上面例子中的Message也可以通过如下方法定义：

```
class Message(typing.TypedDict, total=False):
    uid: int
    time: time.struct_time
    content: str
```

这种方式clsname参数被类名代替，dict参数被类变量标注代替，而total参数则被类参数列表中的关键字total代替（省略则取默认值True）。

Python 3.11给typing模块增加了两个结构——Required和NotRequired。它们的语法分别为：

```
typing.Required[type]
typing.NotRequired[type]
```

它们总是在传入typing.TypedDict()的dict参数的字典中使用，以更精细地表明一个类属性是必须的/可选的。

下面的例子用Required结构表明Message类中的uid和content属性是必须的，而time属性是可选的：

```
$ python3

>>> import time, typing
>>> Message = typing.TypedDict("Message", {"uid": typing.Required[int],
"time": time.struct_time, "content": typing.Required[str]}, total=False)
>>> Message.__total__
False
>>> Message.__required_keys__
frozenset({'content', 'uid'})
>>> Message.__optional_keys__
frozenset({'time'})
>>>
```

该例子还说明了通过typing.TypedDict()创建的类型具有如下三个属性：

- __total__：total参数被传入的布尔值。
- __required_keys__：该类必须的类属性。
- __optional_keys__：该类可选的类属性。

因此当我们获得一个typing.TypedDict()创建的类型时，可以通过这三个属性逆推它在被创建时给typing.TypedDict()传入的实际参数。

下面的例子用NotRequired结构达到了相同的目的，并创建了一些Message对象：

```
import typing
import time


class Message(typing.TypedDict):
    uid: int
    time: typing.NotRequired[time.struct_time]
    content: str


print(f"Message.__total__ == {Message.__total__}")
print(f"Message.__required_keys__ == {Message.__required_keys__}")
print(f"Message.__optional_keys__ == {Message.__optional_keys__}")

#该赋值语句不会报错。
msg1 = Message(uid=0, content="Hello")
#该赋值语句不会报错。
msg2 = Message(uid=0, time=time.localtime(), content="Hello")
#该赋值语句会因却缺少content属性而报错。
msg3 = Message(uid=0, time=time.localtime())
#该赋值语句会因却缺少uid属性而报错。
msg4 = Message(content="Hello")
#该赋值语句会因却多出了data属性而报错。
msg5 = Message(uid=0, content="Hello", data=0)
```

将上述代码保存为typing2.py，首先通过如下命令行验证Message与在前一个例子中相同：

```
$ python3 typing2.py
Message.__total__ == True
Message.__required_keys__ == frozenset({'content', 'uid'})
Message.__optional_keys__ == frozenset({'time'})
```

然后通过如下命令行验证静态类型检查的结果如预期：

```
$ mypy typing2.py
typing2.py:20: error: Missing key "content" for TypedDict
"Message"  [typeddict-item]
typing2.py:22: error: Missing key "uid" for TypedDict
"Message"  [typeddict-item]
typing2.py:24: error: Extra key "data" for TypedDict "Message"  [typeddict-
item]
Found 3 errors in 1 file (checked 1 source file)
```

---

typing中的Literal结构使我们可以指定被赋予的值在指定的字面量中选择，其语法为：

```
typing.Literal[literal, ...]
```

方括号内的部分为一个字面值列表。下面的例子表示函数reader()只能以“r”或“rb”方式打开指定的文件：

```
def reader(file: str, mode: Literal["r", "rb"]) -> str:
    ...
```

需要注意的是，如果给类型注解为Literal结构的变量赋予一个不可哈希对象，则在静态类型检查的过程中会报错。

除了特殊结构之外，typing还定义了一些特殊类型。在Python 3.10及之前的版本中只有两个特殊类型：

- typing.Any：可以匹配任何类型。
- typing.NoReturn：不能匹配任意类型。

从集合论的角度讲，typing.Any代表全集，以之为类型注解说明不对类型做任何限制；而typing.NoReturn则代表空集，以之为类型注解的场合仅见于函数的返回值，表示该函数永远不会返回，下面是一个例子：

```
def stop_immediately() -> typing.NoReturn:
    raise Exception
```

当一个函数的形式参数或返回值，以及一个变量不具有类型注解时，可以认为它们的类型注解默认为typing.Any。

Python 3.11增加了三个特殊类型：

- typing.Never：表示不该被调用的函数。
- typing.Self：用于避免使用前向引用。
- typing.LiteralString：用于抵御SQL注入攻击。

typing.Never是typing.NoReturn的扩展，还可以被用于作为形式参数的类型注解，以表明该函数永远不该被调用，下面是一个例子：

```
    def never_stop(arg: typing.Never) -> typing.Never:
        while True:
            pass
```

typing.Self是为了不再使用前向引用而被引入的，在类体中代表该类体创建的类。当使用typing.Self时，annotation3.pyi中的Pocker.__eq__的定义应该被改写为：

```
    @typing.no_type_check
    def __eq__(self, other: typing.Self): ...
```

Pocker.__lt__的定义应该被改写为：

```
    def __lt__(self, other: typing.Self): ...
```

Draw.compare()的定义应该被改写为：

```
    def compare(self, other: typing.Self) -> int: ...
```

typing.LiteralString是为了抵御SQL注入攻击而被引入的，表示只接受一个字符串字面值，不接受str类型的对象。下面是一个例子：

```
    def exec(sql: typing.LiteralString): ...
```

当静态类型检查工具针对本节介绍的类型注解进行检查时，使用的是“类型签名（type signatures）”方法，即检查被赋予的值的类型是否是类型注解指定的类型的“子类型（subtypes）”。对于内置类型和类来说，如果A是B的__mro__引用的元组的元素（即A是B的基类），那么B就是A的子类型。容易推知，任何类型都是其自身的子类型。

此外，对于bool、int、float和complex，以及fractions模块引入的Fraction，虽然它们中只有bool是int的子类，但numbers模块中的抽象基类注册了它们，存在如下子类型链条：bool → int → Fraction → float → complex。然而decimal模块引入的Decimal不在这个链条上：Decimal是numbers.Number的子类，却不是numbers.Complex的子类。

特别的，bytearray被视为bytes的子类型。

对于特殊类型，则有：

- 任何类型都是typing.Any的子类型。
- 任何类型都不是typing.Never的子类型。
- typing.Self的子类型是它代表的类，及该类派生的子类。

Literal结构和typing.LiteralString则比较特殊，前者检查的其实是值本身而非类型，后者则指定字符串字面值。

不论是上述哪种情况，相应类型注解指定的子类型都属于“名义子类型（nominal subtypes）”。

15-5. 类型理论——鸭子类型

（标准库：typing）

有些时候，我们关心的是被赋予的值是否具有期望的属性，而不关心该值的类型签名。在Python中，我们把期望的属性集合抽象为“协议（protocols）”，如果某个对象具有该属性集合中的所有属性，则称该对象实现了该协议。举例来说，我们把__enter__和__exit__抽象为“上下文管理器协议”，因此任何上下文管理器都实现了上下文管理器协议。类似的，我们把__iter__和__next__抽象为“迭代器协议”，因此任何迭代器都实现了迭代器协议。显然，numbers模块和collections.abc模块定义的每个抽象基类也都可以被抽象为一种协议。

为解决上述问题，必须增加一些类型注解，以指示静态类型检查工具检查一个值是否满足某种协议。这样的类型注解指定的子类型是虚拟的，被称为“结构化子类型（structural subtypes）”。这种技术又被称为“鸭子类型（duck typing）”，源自詹姆斯·莱利提出的鸭子测试：“当看到一只鸟走起来像鸭子、游泳起来像鸭子、叫起来也像鸭子，那么这只鸟就可以被称为鸭子。”

typing提供了Protocol类来支持创建静态鸭子类型。Protocol类以typing._ProtocolMeta为元类，其派生出的子类就是静态鸭子类型。针对Protocol类的子类进行静态类型检查时，会检查值是否具有该类的所有函数属性（鸭子类型不能具有非函数属性），如果是则认为该值的类型是该类的结构化子类型。下面是一个例子：

```
import typing


#定义一个静态鸭子类型Container。
class Container(typing.Protocol):
    def __len__(self) -> int: ...

    def __contains__(self, ele) -> bool: ...


#Example1实现了Container代表的协议。
class Example1():
    def __len__(self):
```

```
            return 0

        def __contains__(self, ele):
            return False


    #Example2的__contains__不符合Container的要求，因此没有实现Container代表的协议。
    class Example2():
        def __len__(self):
            return 0

        def __contains__(self):
            return False


    #声明变量a应该实现Container代表的协议。
    a: Container

    #该赋值语句不会报错，因为tuple实现了Container代表的协议。
    a = (1, 2, 3)
    #该赋值语句不会报错，因为str实现了Container代表的协议。
    a = "abc"
    #该赋值语句会报错，因为int没有实现Container代表的协议。
    a = 0
    #该赋值语句不会报错，因为Example1实现了Container代表的协议。
    a = Example1()
    #该赋值语句会报错，因为Example2没有实现Container代表的协议。
    a = Example2()
```

请将上述代码保存为typing3.pyi，然后通过如下命令行验证：

```
    $ mypy typing3.pyi
    typing3.pyi:37: error: Incompatible types in assignment (expression has
type "int", variable has type "Container")  [assignment]
    typing3.pyi:41: error: Incompatible types in assignment (expression has
type "Example2", variable has type "Container")  [assignment]
    typing3.pyi:41: note: Following member(s) of "Example2" have conflicts:
    typing3.pyi:41: note:     Expected:
    typing3.pyi:41: note:         def __contains__(self, Any, /) -> bool
    typing3.pyi:41: note:     Got:
    typing3.pyi:41: note:         def __contains__(self) -> Any
    Found 2 errors in 1 file (checked 1 source file)
```

Protocol类的子类默认只支持静态类型检查，但如果通过@typing.runtime_checkable装饰器被装饰，则还支持动态类型检查，因而成为一个真正的鸭子类型。这样我们就可以通过issubclass()和isinstance()来判断一个类或对象是否实现了某个协议。但要注意，针对鸭子类型进行动态类型检查时，只会检查属性名是否匹配，不会检查属性的形式参数和返回值是否匹配。请看下面的例子：

```
    import typing


    #定义一个鸭子类型Container。
    @typing.runtime_checkable
    class Container(typing.Protocol):
        def __len__(self) -> int:
```

```
            pass

        def __contains__(self, ele) -> bool:
            pass


    class Example1():
        def __len__(self):
            return 0

        def __contains__(self, ele):
            return False


    class Example2():
        def __len__(self):
            return 0

        def __contains__(self):
            return False
```

请将上述代码保存为typing4.py，然后通过如下命令行和语句验证：

```
$ python3 -i typing4.py

>>> issubclass(tuple, Container)
True
>>> isinstance((1, 2, 3), Container)
True
>>> issubclass(str, Container)
True
>>> isinstance("abc", Container)
True
>>> issubclass(int, Container)
False
>>> isinstance(0, Container)
False
>>> issubclass(Example1, Container)
True
>>> isinstance(Example1(), Container)
True
>>> issubclass(Example2, Container)
True
>>> isinstance(Example2(), Container)
True
>>>
```

注意在进行动态类型检查时，Example2被视为实现了Container代表的协议，因为只检查了属性名。

typing已经定义了若干鸭子类型（都被@typing.runtime_checkable装饰过），被总结在表15-3中。

表15-3. 预定义鸭子类型

类型	相关属性	含义
typing.SupportsBytes	__bytes__	可被视为字节串。
typing.SupportsComplex	__complex__	可被视为复数。
typing.SupportsFloat	__float__	可被视为浮点数。
typing.SupportsInt	__int__	可被视为整数。
typing.SupportsIndex	__index__	可被视为索引。
typing.SupportsAbs	__abs__	支持绝对值运算。
typing.SupportsRound	__round__	支持舍入运算。

最后强调一点，抽象基类与鸭子类型存在逻辑上的联系，但并不是鸭子类型。

15-6. 类型理论——泛型

（语言参考手册：3.3.5）
（标准库：内置类型、types、typing）

请再次回想Tuple结构的语法，注意该结构中的Type1，Type2，……等在形式上类似于函数的形式参数，只不过需要被传入的“实际参数”必须是类型。这里其实已经包含了泛型编程（generic programming）的核心思想：只能引用类型的变量本身也可以被视为一种类型，但这种类型不是具体的，会因该变量引用不同的类型而改变。我们把这种只能引用类型的变量称为“类型变量（type variables）”，而把含有类型变量的类型称为“泛型（generics）”。

---

typing模块提供了TypeVar类来创建类型变量，其语法为：

```
    class typing.TypeVar(typename[, {covariant=True|
contravariant=True}])
    class typing.TypeVar(typename, bound=type[, {covariant=True|
contravariant=True}])
    class typing.TypeVar(typename, type1, type2, ...[,
{covariant=True|contravariant=True}])
```

三种语法都会返回一个TypeVar对象。但第一种语法创建的对象可以引用任何类型，没有任何限制。第二种语法创建的对象只能引用type以及type的子类型，这种限制被称为“绑定（bound）”。第三种语法创建的对象只能引用参数列表type1, type2, ...中指定的类型（不能少于2个）或它们的子类型，这种限制被称为“约束（constraint）”。

类型变量只能出现在函数标注中，不能出现在变量标注中。下面的例子说明了创建类型变量的三种语法的区别：

```
import typing
from collections.abc import Sequence

#定义一个既没有绑定也没有约束的类型变量。
T1 = typing.TypeVar('T1')
#定义一个有绑定的类型变量。
T2 = typing.TypeVar('T2', bound=Sequence)
#定义一个有约束的类型变量。
T3 = typing.TypeVar('T3', tuple, list)


def a(x: T1) -> T1:
    return x


def b(x: T2) -> T2:
    return x


def c(x: T3) -> T3:
    return x


#该函数调用不会报错，因为T1对类型无限制。
a((0, 10))
#该函数调用不会报错，因为tuple是T2的绑定的子类型。
b((0, 10))
#该函数调用不会报错，因为tuple在T3的约束中。
c((0, 10))
#该函数调用不会报错，因为T1对类型无限制。
a([0, 10])
#该函数调用不会报错，因为list是T2的绑定的子类型。
b([0, 10])
#该函数调用不会报错，因为list在T3的约束中。
c([0, 10])
#该函数调用不会报错，因为T1对类型无限制。
a(range(0, 10))
#该函数调用不会报错，因为range是T2的绑定的子类型。
b(range(0, 10))
#该函数调用会报错，因为range不在T3的约束中。
c(range(0, 10))
#该函数调用不会报错，因为T1对类型无限制。
a(0)
#该函数调用会报错，因为int不是T2的绑定的子类型。
b(0)
#该函数调用会报错，因为int不在T3的约束中。
c(0)
```

请将上述代码保存为typing5.pyi，然后通过如下命令行验证：

```
$ mypy typing5.pyi
typing5.pyi:41: error: Value of type variable "T3" of "c" cannot be "range"
[type-var]
typing5.pyi:45: error: Value of type variable "T2" of "b" cannot be
"int"  [type-var]
```

```
      typing5.pyi:47: error: Value of type variable "T3" of "c" cannot be
"int"  [type-var]
      Found 3 errors in 1 file (checked 1 source file)
```

该例子还说明了当作为类型注解时，既没有绑定也没有约束的类型变量等同于typing.Any。

在上面的例子中，以三种语法创建类型变量时都既没有使用关键字covariant也没有使用关键字contravariant。这两个关键字有什么作用呢？为了理解它们，必须先讨论泛型的“协变性（covariance）”、“逆变性（contravariance）”和“不变性（invariance）”。

我们已经知道，泛型就是包含类型变量的类型，当每个类型变量引用的类型被确定下来时，该泛型代表的类型也就确定下来。从数学的角度，假设$\mathbf{T}$代表所有类型形成的集合，那么泛型是定义域为$\mathbf{T}$的n维笛卡尔积、值域为$\mathbf{T}$的一个函数Φ：$\mathbf{T}^n \to \mathbf{T}$。从计算机科学的角度，泛型是一个类型生成器G = TypeGenerator(T_1, T_2, ..., T_n)。现在让我们考虑泛型G和它所依赖的第i个类型变量T_i之间的关系。设(t_1, t_2, ..., t_i, ..., t_n)和(t_1, t_2, ..., t'_i, ..., t_n)是针对泛型的两个“指派（assignments）”，使得该泛型代表的类型分别为T和T'，而这两个指派只在T_i上有区别，分别取t_i和t'_i。

➤ 我们说泛型G相对于T_i是“协变（covariant）”的，指的是G满足如下条件：只要t'_i是t_i的子类型，就可以断定T'是T的子类型。类比于实数域的函数，我们可以认为协变代表的是单调增，例如f(x) = x，当x增大时f(x)也增大。

➤ 我们说泛型G相对于T_i是“逆变（contravariant）”的，指的是G满足如下条件：只要t'_i是t_i的子类型，就可以断定T是T'的子类型。类比于实数域的函数，我们可以认为逆变代表的是单调减，例如f(x) = -x，当x增大时f(x)减小。

➤ 我们说泛型G相对于T_i是“不变（invariant）”的，指的是G满足如下条件：无法根据t'_i和t_i之间的子类型关系，推断T'和T之间的子类型关系。类比于实数域的函数，我们可以认为不变代表的是既非单调增也非单调减，例如f(x) = x^2，当x增大时f(x)有可能增大也有可能减小。（当然对于T和T'来说，存在三种关系：T'是T的子类型，T是T'的子类型，T和T'相互间均不是对方的子类型。）

typing.TypeVar中的关键字covariant和contravariant只能被设置为True，且两者不能同时存在。如果存在covariant，则说明创建的类型变量是协变的；如果存在contravariant，则说明创建的类型变量是逆变的；如果两个关键字都不存在，则说明创建的类型变量是不变的。在typing5.pyi中创建的所有类型变量都是不变的。

说一个类型变量是协变的、逆变的或不变的，其实是不严谨的。这些概念仅在针对某个泛型的语境中才有意义。如果想定义指定名义子类型的泛型，则只需要让自定义类继承Generic抽象基类，而该抽象基类可以通过如下语法包含任意（但不能是0）个类型变量：

```
typing.Generic[T, ...]
```

如果想定义指定结构化子类型的泛型，则只需要让自定义类同时继承Protocol和Generic，或者简写为：

```
typing.Protocol[T, ...]
```

上述语法等价于“typing.Protocol, typing.Generic[T, ...]”。而在这些代表泛型的类的类体中，通过上述方式包含的类型变量不仅可以用于函数标注，还可以用于变量标注。

那么在通过上述方式创建泛型时，我们该如何判断该设置相关类型变量是协变的、逆变的还是不变的呢？由于类型变量只能出现在函数标注中，所以它们其实只存在3种情况。而由于普通泛型只检查类型签名，鸭子泛型则要检查每个类属性，所以在相同的情况下两者的要求可能不同。具体来说，泛型中的类型变量必须遵守如下规则：

➤ 当该类型变量既被用作形式参数的类型注解，又被用作返回值的类型注解时，不论在普通泛型中还是在鸭子泛型中，它都必须是不变的。
➤ 当该类型变量仅被用作形式参数的类型注解时，则在普通泛型中可以是逆变的或不变的，而在鸭子泛型中只能是逆变的。
➤ 当该类型变量仅被用作返回值的类型注解时，不论在普通泛型中还是在鸭子泛型中，它都不受任何限制，可以是协变的、逆变的和不变的。

下面的例子定义了一个普通泛型NumMapping，它建立在类型变量KT、VT和ST之上：

```
import typing

#代表实数或布尔值的逆变类型变量，它只被用作形式参数的类型注解。
KT = typing.TypeVar('KT', bound=typing.SupportsInt, contravariant=True)

#代表实数或布尔值的不变类型变量，它同时被用作形式参数和返回值的类型注解。
VT = typing.TypeVar('VT', bound=typing.SupportsInt)

#代表实数的协变类型变量，它只被用作返回值的类型注解。
ST = typing.TypeVar('ST', int, float, covariant=True)


#代表实数或布尔值之间映射的泛型。
class NumMapping(typing.Generic[KT, VT, ST]):
    default = False
```

```
    def __init__(self, data: dict[KT, VT]):
        for k, v in data.items():
            setattr(self, str(k), v)

    def __getitem__(self, k: KT) -> VT:
        #这里利用typing.cast()对self.default进行了强制类型转换。
        return getattr(self, str(k), typing.cast(VT, self.default))

    def __setitem__(self, k: KT, v: VT):
        setattr(self, str(k), v)

    def __delitem__(self, k: KT):
        delattr(self, str(k))

    def __repr__(self):
        s = ""
        for k, v in self.__dict__.items():
            if k[0] != "_":
                s += k + ": " + str(v) + ", "
        return "{" + s.strip(", ") + "}"

    __str__ = __repr__

    #计算映射中所有值之和。
    def sum(self) -> ST:
        total = 0
        for k, v in self.__dict__.items():
           if k[0] != "_":
                total += getattr(self, k)
        return total


if __name__ == '__main__':
    #该赋值语句不会报错，因为int在KT的绑定中，而bool,int和float都在VT的绑定中。
    skm1 = NumMapping({1: True, 2: 2, 3: 3.0})
    #该赋值语句会报错，因为str不在KT的绑定中。
    skm2 = NumMapping({'1': True, '2': 2, '3': 3.0})
    #该赋值语句会报错，因为complex不在VT的绑定中。
    skm3 = NumMapping({1: 1j, 2: 2j, 3: 3j})
```

请将上述代码保存为typing6.py，然后先通过如下命令行和语句验证在类型注解中使用泛型依然不影响NumMapping的实例化：

```
$ python3 -i typing6.py

>>> skm1
{1: True, 2: 2, 3: 3.0}
>>> skm2
{1: True, 2: 2, 3: 3.0}
>>> skm3
{1: 1j, 2: 2j, 3: 3j}
>>> skm1[1]
True
>>> skm2[2]
2
>>> skm3[3]
3j
>>> skm1.sum()
6.0
>>> skm2.sum()
```

```
6.0
>>> skm3.sum()
6j
>>>
```

然后再通过如下命令行验证在类型注解中使用泛型同样可以限制值的类型（只不过该限制从一个类型扩展到一个类型范围）：

```
$ mypy typing6.py
typing6.py:53: error: Value of type variable "KT" of "NumMapping" cannot be "str"  [type-var]
typing6.py:55: error: Value of type variable "VT" of "NumMapping" cannot be "complex"  [type-var]
Found 2 errors in 1 file (checked 1 source file)
```

值得一提的是，上面的例子中用到了typing.cast()，其语法为：

typing.cast(*type*, *value*)

其作用是将value的类型强制转换成type。事实上，当Python脚本被解释器执行时该函数没有任何作用，因为它总是原封不动地返回value。该函数仅在静态类型检查时才有意义，作用是使对value的类型检查被跳过。

上面的例子显示了泛型与普通类型相似的点，而下面的例子则显示了泛型与普通类型不同的点：

```
from typing6 import KT, VT, ST, NumMapping

#创建4个NumMapping的泛型别名。
type1 = NumMapping[int, bool, float]
type2 = NumMapping[int, bool, int]
type3 = NumMapping[float, bool, float]
type4 = NumMapping[float, bool, int]

#声明四个以不同泛型别名为类型注解的变量。
a: type1
b: type2
c: type3
d: type4

#验证不同泛型别名允许的子类型范围。
if __name__ == '__main__':
    a = type4({1.0: True, 0.0: False})
    a = type3({1.0: True, 0.0: False})
    a = type2({1: True, 0: False})
    a = type1({1: True, 0: False})

    b = type4({1.0: True, 0.0: False})
    b = type3({1.0: True, 0.0: False})    #报错。
    b = type2({1: True, 0: False})
    b = type1({1: True, 0: False})    #报错。
```

```
        c = type4({1.0: True, 0.0: False})
        c = type3({1.0: True, 0.0: False})
        c = type2({1: True, 0: False})    #报错。
        c = type1({1: True, 0: False})    #报错。

        d = type4({1.0: True, 0.0: False})
        d = type3({1.0: True, 0.0: False})    #报错。
        d = type2({1: True, 0: False})    #报错。
        d = type1({1: True, 0: False})    #报错。
```

请将上述代码保存为typing7.py，然后通过如下命令行验证：

```
    $ mypy typing7.pyi
    typing6.py:53: error: Value of type variable "KT" of "NumMapping" cannot be
"str"  [type-var]
    typing6.py:55: error: Value of type variable "VT" of "NumMapping" cannot be
"complex"  [type-var]
    typing7.py:23: error: Incompatible types in assignment (expression has type
"NumMapping[float, bool, float]", variable has type "NumMapping[int, bool, int]")
[assignment]
    typing7.py:25: error: Incompatible types in assignment (expression has type
"NumMapping[int, bool, float]", variable has type "NumMapping[int, bool, int]")
[assignment]
    typing7.py:29: error: Incompatible types in assignment (expression has type
"NumMapping[int, bool, int]", variable has type "NumMapping[float, bool, float]")
[assignment]
    typing7.py:30: error: Incompatible types in assignment (expression has type
"NumMapping[int, bool, float]", variable has type "NumMapping[float, bool,
float]")  [assignment]
    typing7.py:33: error: Incompatible types in assignment (expression has type
"NumMapping[float, bool, float]", variable has type "NumMapping[float, bool,
int]")  [assignment]
    typing7.py:34: error: Incompatible types in assignment (expression has type
"NumMapping[int, bool, int]", variable has type "NumMapping[float, bool, int]")
[assignment]
    typing7.py:35: error: Incompatible types in assignment (expression has type
"NumMapping[int, bool, float]", variable has type "NumMapping[float, bool, int]")
[assignment]
    Found 9 errors in 2 files (checked 1 source file)
```

从上面的结果可以得出两个很重要的结论。第一个结论是泛型确实是一个类型生成器，但将一个指派应用于泛型的方法与函数调用不同，需要采用下面的格式：

```
    Generic[type1, type2, ..., typeN]
```

这会导致返回一个具体的类型，被称为“泛型别名（generic aliases）”，例如上面例子中的type1~type4。

第二个结论是，要想从泛型派生出来两个类型typeA和typeB，并使的typeA是typeB子类型，则必须同时满足下面三个条件：

- 对于该泛型的任意不变类型变量，typeA和typeB必须取相同类型。
- 对于该泛型的任意协变类型变量，typeA取的类型必须是typeB取的类型的子类型。
- 对于该泛型的任意逆变类型变量，typeB取的类型必须是typeA取的类型的子类型。

对于上面的例子来说，VT、ST和KT分别属于条件1、2和3中的情况，因此type1～type4之间的关系如图15-4所示。

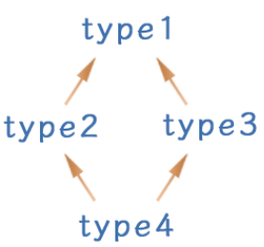

图15-4. typing6中各泛型别名之间的关系

在Python中，泛型别名的类型是通过内置类型types.GenericAlias来表示的。然而按照上面的方式创建的类型别名并不是内置类型，而是一个以typing._GenericAlias为元类的类。types.GenericAlias和typing._GenericAlias并不相等，但它们的实例具有相同的属性：

- __origin__：引用该泛型别名所属的泛型。
- __args__：引用一个元组，代表该泛型别名被创建时使用的指派。
- __parameters__：引用一个元组，仅当该泛型别名被创建时使用的指派包含类型变量时才不为空。

下面沿用typing7.py来验证泛型别名的上述三个属性：

```
$ python3 -i typing7.py

>>> type(type1)
<class 'typing._GenericAlias'>
>>> type1.__origin__
<class 'typing6.NumMapping'>
>>> type1.__args__
(<class 'int'>, <class 'bool'>, <class 'float'>)
>>> type1.__parameters__
()
>>>
>>> type(type2)
<class 'typing._GenericAlias'>
>>> type2.__origin__
<class 'typing6.NumMapping'>
>>> type2.__args__
(<class 'int'>, <class 'bool'>, <class 'int'>)
>>> type1.__parameters__
()
>>>
>>> type(type3)
<class 'typing._GenericAlias'>
```

```
>>> type3.__origin__
<class 'typing6.NumMapping'>
>>> type3.__args__
(<class 'float'>, <class 'bool'>, <class 'float'>)
>>> type1.__parameters__
()
>>>
>>> type(type4)
<class 'typing._GenericAlias'>
>>> type4.__origin__
<class 'typing6.NumMapping'>
>>> type4.__args__
(<class 'float'>, <class 'bool'>, <class 'int'>)
>>> type1.__parameters__
()
>>>
>>> import types, typing
>>> typing._GenericAlias == types.GenericAlias
False
>>>
```

上面的例子只说明了如何创建和使用普通泛型。下面的例子创建一个鸭子泛型，并用它来实现typing7.py中的类似验证：

```
from typing6 import KT, VT, ST, NumMapping, typing


#代表实数或布尔值之间映射的鸭子泛型。
class NumMappingProtocol(typing.Protocol[KT, VT, ST]):
    def __init__(self, data: dict[KT, VT]): ...

    def __getitem__(self, k: KT) -> VT: ...

    def __setitem__(self, k: KT, v: VT): ...

    def __delitem__(self, k: KT): ...

    def sum(self) -> ST: ...


#创建4个NumMapping的泛型别名。
type1 = NumMapping[int, bool, float]
type2 = NumMapping[int, bool, int]
type3 = NumMapping[float, bool, float]
type4 = NumMapping[float, bool, int]

#创建4个NumMappingProtocol的泛型别名。
protocol1 = NumMappingProtocol[int, bool, float]
protocol2 = NumMappingProtocol[int, bool, int]
protocol3 = NumMappingProtocol[float, bool, float]
protocol4 = NumMappingProtocol[float, bool, int]

#声明四个以不同泛型别名为类型注解的变量。
a: protocol1
b: protocol2
c: protocol3
d: protocol4
```

```
#验证不同泛型别名允许的子类型范围。
if __name__ == '__main__':
    a = type4({1.0: True, 0.0: False})
    a = type3({1.0: True, 0.0: False})
    a = type2({1: True, 0: False})
    a = type1({1: True, 0: False})

    b = type4({1.0: True, 0.0: False})
    b = type3({1.0: True, 0.0: False})    #报错。
    b = type2({1: True, 0: False})
    b = type1({1: True, 0: False})    #报错。

    c = type4({1.0: True, 0.0: False})
    c = type3({1.0: True, 0.0: False})
    c = type2({1: True, 0: False})    #报错。
    c = type1({1: True, 0: False})    #报错。

    d = type4({1.0: True, 0.0: False})
    d = type3({1.0: True, 0.0: False})    #报错。
    d = type2({1: True, 0: False})    #报错。
    d = type1({1: True, 0: False})    #报错。
```

请将上述代码保存为typing8.pyi，然后可以通过mypy验证它的输出与typing7.py相同。

鸭子泛型与普通泛型的区别主要是以下两点：

➤ 普通泛型和以它创建的泛型别名可以被实例化，而鸭子泛型和以它创建的泛型别名不能被实例化。
➤ 普通泛型和以它创建的泛型别名不支持动态类型检查（即不能作为issubclass()和isinstance()的参数）。而经过@typing..runtime_checkable装饰后，鸭子泛型和以它创建的泛型别名支持动态类型检查。

值得说明的是，当实例化一个普通泛型时，会根据给__init__传入的实际参数来判断应如何给相关类型变量选择类型。

至此我们已经有了泛型的基本概念。容易推知，前面介绍的Tuple结构其实是一个泛型。这可以通过如下方法来验证：

```
$ python3

>>> import typing
>>> T = typing.Tuple[int, float]
>>> type(T)
<class 'typing._GenericAlias'>
>>>
```

而Literal结构则是一个特殊泛型，因为使用它时给出的指派包含的是字面量而非类型，这可以通过如下方法来验证：

```
>>> T = typing.Literal[0, None, 'abc']
>>> type(T)
<class 'typing._LiteralGenericAlias'>
```

TypedDict本身不是泛型，但可以支持泛型，请看下面的例子：

```
from typing import TypeVar, TypedDict, Generic

T = TypeVar('T')


#声明一个属于泛型的TypedDict类型。
class Group(TypedDict, Generic[T]):
    leader: T
    members: list[T]


class Person():
    pass


#创建泛型别名。
PersonGroup = Group[Person]

p1 = Person()
p2 = Person()

pg1 = PersonGroup(leader=p1, members=[p1, p2])
pg2 = PersonGroup(leader=p1, members=p2)    #报错。

print(pg1)
print(pg2)
```

请将上述代码保存为typing9.py，先通过如下命令行和语句验证：

```
$ python3 -i typing9.py
{'leader': <__main__.Person object at 0x104e58790>, 'members':
[<__main__.Person object at 0x104e58790>, <__main__.Person object at
0x104e58810>]}
{'leader': <__main__.Person object at 0x104e58790>, 'members':
<__main__.Person object at 0x104e58810>}
>>> type(Group)
<class 'typing._TypedDictMeta'>
>>> type(PersonGroup)
<class 'typing._GenericAlias'>
>>>
```

再通过如下命令行验证：

```
      $ mypy typing9.py
      typing9.py:23: error: Incompatible types (expression has type "Person",
TypedDict item "members" has type "List[Person]")  [typeddict-item]
      Found 1 error in 1 file (checked 1 source file)
```

typing.AnyStr是typing模块预定义的一个类型变量，它就等价于TypeVar('AnyStr', str, bytes)，意味着可以接收属于str或bytes，以及它们派生出的子类（例如bytearray）的对象。下面的例子说明了该类型变量的用法：

```
      from typing import Tuple, AnyStr


      def record(name: AnyStr, data: AnyStr) -> Tuple[AnyStr, AnyStr]:
          return (name, data)


      record("abc", "xyz")
      record(b"abc", b"xyz")
      record(bytearray(b"abc"), bytearray(b"xyz"))
      record("abc", b"xyz")    #报错。
      record("abc", bytearray(b"xyz"))    #报错。
      record(b"abc", bytearray(b"xyz"))
```

请将上述代码保存为typing10.pyi，然后通过如下命令行验证：

```
      $ mypy typing10.pyi
      typing10.pyi:11: error: Value of type variable "AnyStr" of "record" cannot
be "Sequence[object]"  [type-var]
      typing10.pyi:12: error: Value of type variable "AnyStr" of "record" cannot
be "Sequence[object]"  [type-var]
      Found 2 errors in 1 file (checked 1 source file)
```

该结果说明，name参数和data参数必须或者同时为str或其子类，或者同时为bytes或其子类。注意如果我们自定义了类型变量TypeVar('T', str, bytes)，那用T代替上面例子中的每一处AnyStr（除了import语句）也可以达到相同的效果。如果同一类型变量在一个函数标注（以及一个泛型）中多次出现，则说明该函数（泛型）内部存在类型依赖的情况，即某些形式参数和/或返回值的类型必须相同。

现在让我们把注意力放在Tuple结构上，并思考这样一个问题：如何表示一个值的类型可以是具有任意多个元素，且这些元素具有任意类型的元组？你会发现利用前面介绍的工具无法达到这一目的。而这个问题其实相当普遍，例如*args就需要被传入一个这样的元组，而函数的返回值也可能是一个这样的元组。（*kwargs不存在这一问题，它的类型注解通常为一个由typing.TypedDict()创建的类型。）为了解决上述问题，Python 3.11引入了typing.TypeVarTuple类和typing.Unpack操作符。

typing.TypeVarTuple被用来创建类型变量元组，其实例化语法为：

```
class typing.TypeVarTuple(typetuplename)
```

这会创建一个TypeVarTuple对象，它代表任意多（可以是0）个类型形成的元组。

当一个泛型包含类型变量元组时，必须在其前面添加“*”，以表示将该类型变量元组展开。然而这一语法特征是Python 3.11引入的，为了兼容Python 3.10及之前的版本，需要用typing.Unpack来代替*，其语法为：

```
typing.Unpack[typetuplename]
```

一个泛型可以同时包含任意多个类型变量元组的展开，以及任意多个类型变量。

下面的例子说明了如何使用类型变量元组：

```
from typing import cast, Tuple, TypeVar, TypeVarTuple, Unpack

T = TypeVar('T', bound=float)

#定义类型变量元组。
Ts = TypeVarTuple('Ts')

#定义一个代表元组的类型，该元组至少有一个元素，且第一元素必须是浮点数。
TT = Tuple[T, *Ts]
#兼容Python 3.10的写法。
#TT = Tuple[T, Unpack[Ts]]


def sum(data: TT) -> T:
    x = cast(T, 0)
    for y in data:
        x += y
    return x


if __name__ == "__main__":
    print(sum((0, 1.5, True)))
    print(sum(tuple()))    #报错。
    print(sum((1j, 2j)))    #报错。
```

请将上述代码保存为typing11.py，先通过如下命令行和语句验证：

```
$ python3 -i typing11.py
2.5
0
3j
>>> type(Ts)
<class 'typing.TypeVarTuple'>
```

```
>>> type(TT)
<class 'typing._GenericAlias'>
>>> TT.__origin__
<class 'tuple'>
>>> TT.__args__
(~T, *Ts)
>>> TT.__parameters__
(~T, Ts)
>>>
```

注意该例子中Tuple结构的指派包含一个类型变量和一个类型变量元组的展开，所以它产生的泛型别名本质上依然是个泛型，其__parameters__属性不再引用空元组。

遗憾的是，在本书被编写时的mypy版本（即0.991）依然不支持typing.TypeVarTuple和typing.Unpack。事实上这个语法特性还不完善，比如类型变量元组无法像类型变量那样指定绑定或约束，以及协变、逆变或不变。

最后，typing模块还给出了Type结构，其语法为：

```
typing.Type[t_co]
```

其中t_co可以是如下几种情况：

- 一个类的类名：此时该结构表示该类和该类派生的所有子类。
- 一个类型变量：此时该结构依然是泛型。
- 一个若干类的类名和若干类型变量构成的联合类型（会在后面介绍）：此时如果联合类型中没有类型变量，则表示所有相关类及它们派生的所有子类；否则依然是泛型。
- typing.Any：此时该结构退化为元类type。

注意t_co中的“_co”是为了表明这是一个协变类型变量，因为若A是B的子类，Type[A]显然是Type[B]的子类型。

当以Type结构作为类型注解时，表示应该传入或返回某个类对象。下面的例子说明了如何使用Type结构：

```
import typing


#所有类的基类。
class PointBase(): ...


#直接派生自PointBase。
class Point3D(PointBase): ...
```

```
#直接派生自PointBase。
class PointMove(PointBase): ...


#直接派生自PointBase。
class PointShow(PointBase): ...


#派生自Point3D。
class Point3DMove(Point3D): ...


#T代表Point3D和它的子类。
T = typing.Type[Point3D]


def create_point(cls: T): ...


if __name__ == "__main__":
    create_point(PointBase)    #报错。
    create_point(Point3D)
    create_point(PointMove)    #报错。
    create_point(PointShow)    #报错。
    create_point(Point3DMove)
```

请将上述代码保存为typing12.pyi，然后通过如下命令行验证：

```
$ mypy typing12.pyi
typing12.pyi:32: error: Argument 1 to "create_point" has incompatible type
"Type[PointBase]"; expected "Type[Point3D]"  [arg-type]
typing12.pyi:34: error: Argument 1 to "create_point" has incompatible type
"Type[PointMove]"; expected "Type[Point3D]"  [arg-type]
typing12.pyi:35: error: Argument 1 to "create_point" has incompatible type
"Type[PointShow]"; expected "Type[Point3D]"  [arg-type]
Found 3 errors in 1 file (checked 1 source file)
```

至此我们已经讨论完了自定义泛型的所有技巧。值得强调的是，Python中的很多内置类型和预定义抽象基类本质上都是泛型。举例来说，当内置类型tuple、list、set和frozenset的元素的类型都相同时，其类型可以通过如下语法来精确表示：

```
collection[type]
```

其中collection是容器类型的名称，而type是任意类型。下面是一些例子：

```
#由整数形成的元组。
tuple[int]
```

```
#由以type为元类的类对象形成的列表。
list[type]

#由列表形成的集合。
set[list]

#以由布尔值形成的元组为元素的凝固集合。
frozenset[tuple[bool]]
```

而当映射的键和值的类型都相同时，其类型可以通过如下语法来精确表示：

```
mapping[ktype, vtype]
```

其中ktype是键的类型，vtype是值的类型，例如：

```
#由以字符串为键以浮点数为值的键值对形成的字典。
dict[str, float]
```

需要强调的是，通过上述方法获得的泛型别名的类型是types.GenericAlias。typing7.py已经说明，types.GenericAlias和typing._GenericAlias是不同的。事实上，我们也可以通过直接调用types.GenericAlias来创建泛型别名，因为它本质上是一个类，实例化语法为：

```
class types.GenericAlias(t_origin, t_args)
```

其中t_origin参数是一个非形参化泛型（即内置类型和预定义抽象基类中的泛型），而t_args参数是一个类型构成的元组。可以通过下面的命令行和语句验证两种方法得到的泛型别名是相同的：

```
$ python3

>>> import types
>>> T1 = tuple[int]
>>> T2 = types.GenericAlias(tuple, (int,))
>>> T1 == T2
True
>>> T3 = list[type]
>>> T4 = types.GenericAlias(list, (type,))
>>> T3 == T4
True
>>> T5 = dict[str, float]
>>> T6 = types.GenericAlias(dict, (str, float))
>>> T5 == T6
True
>>> T7 = set[list]
>>> T8 = types.GenericAlias(set, (list,))
>>> T7 == T8
True
>>> T9 = frozenset[tuple[bool]]
```

```
>>> T10 = types.GenericAlias(frozenset, (tuple[bool],))
>>> T9 == T10
True
>>>
```

请继续通过下面的语句验证，types.GenericAlias对象也具有__origin__、__args__和__parameters__属性：

```
>>> type(T1)
<class 'types.GenericAlias'>
>>> T1.__origin__
<class 'tuple'>
>>> T1.__args__
(<class 'int'>,)
>>> T1.__parameters__
()
>>> type(T3)
<class 'types.GenericAlias'>
>>> T3.__origin__
<class 'list'>
>>> T3.__args__
(<class 'type'>,)
>>> T3.__parameters__
()
>>> type(T5)
<class 'types.GenericAlias'>
>>> T5.__origin__
<class 'dict'>
>>> T5.__args__
(<class 'str'>, <class 'float'>)
>>> T5.__parameters__
()
>>> type(T7)
<class 'types.GenericAlias'>
>>> T7.__origin__
<class 'set'>
>>> T7.__args__
(<class 'list'>,)
>>> T7.__parameters__
()
>>> type(T9)
<class 'types.GenericAlias'>
>>> T9.__origin__
<class 'frozenset'>
>>> T9.__args__
(tuple[bool],)
>>> T9.__parameters__
()
>>>
```

之所以存在两种代表泛型别名的类，是因为types.GenericAlias是由types模块提供的，针对的是类型对象；而typing._GenericAlias是由typing模块提供的，针对的是类对象。两者的区别属于Python实现的细节，对于用户来说完全没有意义。为了能让Python用户以一种统一的方式看待泛型别名，typing模块为所有types.GenericAlias对象都提供了“泛型具象（generic concretes）”，被总结在附录 V中。这使得每个types.GenericAlias对象都可以被等价视为某个属于typing._GenericAlias或其子类的对象。

typing模块还提供了函数typing.get_origin()和typing.get_args()来分别取得一个泛型别名的__origin__和__args__引用的对象，语法为：

```
typing.get_origin(tp)
typing.get_args(tp)
```

其中tp参数需被传入一个泛型别名，可以是types.GenericAlias对象，也可以是属于typing._GenericAlias或其子类的对象。

需要强调的是，对于泛型具象来说，不论是直接访问__origin__或__args__，还是调用typing.get_origin()或typing.get_args()，都会先将其转化为typing._GenericAlias对象。下面的例子说明了这点：

```
$ python3

>>> import typing
>>> T1 = typing.List[float]
>>> T1.__origin__
<class 'list'>
>>> typing.get_origin(T1)
<class 'list'>
>>> T1.__args__
(<class 'float'>,)
>>> typing.get_args(T1)
(<class 'float'>,)
>>>
>>> T2 = typing.Dict[str, int]
>>> T2.__origin__
<class 'dict'>
>>> typing.get_origin(T2)
<class 'dict'>
>>> T2.__args__
(<class 'str'>, <class 'int'>)
>>> typing.get_args(T2)
(<class 'str'>, <class 'int'>)
>>>
```

在本节的最后将简要介绍一下在Python中实现泛型的机制。从上面的例子可以看出，通过一个泛型创建泛型别名的语法，与访问一个容器类型的元素的语法类似。事实上，创建泛型别名时会调用相应泛型的__class_getitem__属性，其语法为：

```
Generic.__class_getitem__(cls, key)
```

其中cls是泛型的类名，key则是方括号内的类型列表。事实上，只要一个类是泛型的充分必要条件是它具有__class_getitem__属性。与__init_subclass__类似，__class_getitem__总是被识别为类方法，即便不使用@classmethod装饰器。

虽然理论上可以自定义__class_getitem__，但这是强烈不推荐的，因为该属性必须返回一个代表泛型别名的对象，即属于typing._GenericAlias或其子类的对象，而我们并不知道该如何创建此类对象。typing.Generic实现了默认的__class_getitem__，自定义泛型总是通过继承typing.Generic来获得__class_getitem__。

就好像对容器类型执行抽取或切片操作时会自动调用容器类型的__getitem__一样，给予泛型一个指派时会自动调用该泛型的__class_getitem__。由于这两种操作都涉及方括号，所以“a[...]”会被按照如下步骤解析：

➤ 步骤一：在type(a)中搜索魔术属性__getitem__，如果找到则调用它，完成解析；否则执行步骤二。
➤ 步骤二：在type(a)中搜索__class_getitem__，如果找到则调用它，完成解析；否则抛出TypeError异常。

由于元类type并不具有__getitem__，所以对于同时具有这两个属性的类，例如tuple和list，“tuple[...]”和“list[]”会导致调用__class_getitem__，而“tuple(...)[...]”和“list(...)[...]”会导致调用__getitem__。然而如果某个自定义元类具有__getitem__，则即便它具有__class_getitem__该属性也会被屏蔽。

需要强调的是，泛型本身是type的实例，可以被看成一个类，因此可以作为issubclass()和isinstance()的第二个参数。但泛型别名是types.GenericAlias或typing._GenericAlias或它们的子类的实例，并不是真正的类，因此不能作为issubclass()和isinstance()的第二个参数。下面的例子说明了这点：

```
$ python3

>>> import typing
>>> T = typing.TypeVar('T')
>>> class G(typing.Generic[T]):
...     pass
...
>>> type(G)
<class 'type'>
>>> G.__mro__
(<class '__main__.G'>, <class 'typing.Generic'>, <class 'object'>)
>>> type(G[int])
<class 'typing._GenericAlias'>
>>> typing._GenericAlias.__mro__
(<class 'typing._GenericAlias'>, <class 'typing._BaseGenericAlias'>, <class
'typing._Final'>, <class 'object'>)
>>> issubclass(int, G)
False
>>> issubclass(int, G[int])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/
typing.py", line 1250, in __subclasscheck__
    raise TypeError("Subscripted generics cannot be used with"
    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

```
    TypeError: Subscripted generics cannot be used with class and instance
checks
    >>>
```

然而，泛型别名像泛型一样可以出现在类参数列表中的。这是因为泛型别名所属类（即types.GenericAlias或typing._GenericAlias及其子类）实现了__mro_entries__。前面已经提到了，由于泛型别名不是元类type的实例，且具有__mro_entries__属性，所以当它出现在类参数列表时会以导致用如下调用的结果替换它：

```
    Generic.__mro_entries__(self, bases)
```

其中bases为类参数列表中的所有直接基类形成的元组。泛型的__mro_entries__会返回形式为“(self.__origin__,)”的元组，也就是以相应泛型代替泛型别名。请看下面的例子：

```
    import typing

    T = typing.TypeVar("T")

    class A(typing.Generic[T]):
        pass

    class B(A[int], list[int]):
        pass

    print(f"B.__bases__ == {B.__bases__}")
    print(f"B.__mro__ == {B.__mro__}")
```

请将上述代码保存为typing13.py，然后通过如下命令行验证：

```
    $ python3 typing13.py
    B.__bases__ == (<class '__main__.A'>, <class 'list'>)
    B.__mro__ == (<class '__main__.B'>, <class '__main__.A'>, <class
'typing.Generic'>, <class 'list'>, <class 'object'>)
```

可见B的类参数列表中的A[int]其实指定了直接基类A，list[int]其实指定了直接基类list。当一个类的类参数列表中包含泛型或泛型别名时，该类的MRO中必然有typing.Generic或typing.Protocol，因此自身也会成为泛型。

15-7. 类型理论——可调用类型的表示

（标准库：typing）

到目前为止，我们给出的所有例子都不涉及可调用对象的类型。然而在第4章讨论匿名函数时，已经给出了以回调函数作为实际参数，以及返回值是函数的例子。装饰器也是以可调用对象为实际参数，并返回另一个可调用对象。在这一节将详细讨论如何表示可调用对象的类型。

表示可调用类型的基本方法是只使用Callable结构，其第一种语法为：

```
Callable[[atype1, ..., atypeN], rtype]
```

这种Callable结构代表的是一个具有N个参数类型分别是atype1到atypeN，以及一个返回对象类型是rtype的可调用对象的类型。根据该可调用对象的具体类型，Callable结构的解读方式存在如下两种：

➤ 当Callable结构对应的可调用对象是类时，atype1到atypeN代表的是直接基类，rtype代表的就是该类本身，而类参数列表中的关键字无法表示。
➤ 当Callable结构对应的可调用对象是函数、匿名函数、生成器函数、协程函数或异步生成器函数时，atype1到atypeN代表的是能基于位置对应的参数的类型，rtype代表的是返回值、括生成器、协程或异步生成器的类型，而仅关键字形式参数的类型无法表示。

由于大多数情况下，回调函数和作为返回值的函数都比较简单，不涉及仅关键字形式参数，所以Callable结构的第一种语法就足以胜任。下面的存根文件bubble_sorts.pyi改编自第4章中的例子bubble_sort.py：

```
      from typing import Protocol, Callable
      from collections.abc import Sequence


      #回调函数的类型为Callable[[int, int], float]。
      def bubble_sort_int(lt: list, compare: Callable[[int, int], float]) ->
list: ...


      #回调函数的类型为Callable[[float, float], float]。
      def bubble_sort_float(lt: list, compare: Callable[[float, float], float])
-> list: ...


      #回调函数的类型为Callable[[str, str], float]。
      def bubble_sort_str(lt: list, compare: Callable[[str, str], float]) ->
list: ...


      if __name__ == "__main__":
          #该函数的类型为Callable[[int, int], bool]。
          def f1(x: int, y: int) -> bool: ...


          #该函数的类型为Callable[[float, float], int]。
          def f2(x: float, y: float) -> int: ...


          #该函数的类型为Callable[[Sequence, Sequence], float]。
```

```
        def f3(x: Sequence, y: Sequence) -> float: ...


        bubble_sort_int(list(), f1)
        bubble_sort_int(list(), f2)
        bubble_sort_int(list(), f3)    #报错。

        bubble_sort_float(list(), f1)    #报错。
        bubble_sort_float(list(), f2)
        bubble_sort_float(list(), f3)    #报错。

        bubble_sort_str(list(), f1)    #报错。
        bubble_sort_str(list(), f2)    #报错。
        bubble_sort_str(list(), f3)
```

请将上述代码保存为typing14.pyi，然后通过如下命令行验证：

```
    $ mypy typing14.pyi
    typing14.pyi:32: error: Argument 2 to "bubble_sort_int" has incompatible
type "Callable[[Sequence[Any], Sequence[Any]], float]"; expected "Callable[[int,
int], float]"  [arg-type]
    typing14.pyi:34: error: Argument 2 to "bubble_sort_float" has incompatible
type "Callable[[int, int], bool]"; expected "Callable[[float, float],
float]"  [arg-type]
    typing14.pyi:36: error: Argument 2 to "bubble_sort_float" has incompatible
type "Callable[[Sequence[Any], Sequence[Any]], float]"; expected
"Callable[[float, float], float]"  [arg-type]
    typing14.pyi:38: error: Argument 2 to "bubble_sort_str" has incompatible
type "Callable[[int, int], bool]"; expected "Callable[[str, str], float]"  [arg-
type]
    typing14.pyi:39: error: Argument 2 to "bubble_sort_str" has incompatible
type "Callable[[float, float], int]"; expected "Callable[[str, str],
float]"  [arg-type]
    Found 5 errors in 1 file (checked 1 source file)
```

注意该结果说明Callable结构中的atype1到atypeN都是逆变的，而rtype则是协变的。

Callable结构的第二种语法为：

```
    Callable[..., rtype]
```

这种Callable结构代表的是对参数没有类型限制，返回对象类型是rtype的可调用对象。这意味着Callable结构可以表示包含仅关键字形式参数的函数，只不过此时无法对参数的类型做任何限制。下面的例子说明了如何使用这种Callable结构：

```
    from typing import Callable, Any
    import collections.abc


    #该函数可以调用任何返回值类型为int的函数，并返回获得的返回值。   在正式执行之前，会先显
    # 示即将执行的函数调用。
    def verbose(func: Callable[..., int], *args: Any, **kwargs: Any) -> int:
```

```
        params = ""
        for v in args:
            params += str(v) + ", "
        for k, v in kwargs.items():
            params += str(k) + "=" + str(v)+", "
        print(f"executing {func.__name__}({params.strip(', ')})")
        return func(*args, **kwargs)


    if __name__ == "__main__":
        #该函数的类型为Callable[[], bool]。
        def f1() -> bool:
            return True


        #该函数的类型为Callable[[int, int], int]。
        def f2(x: int, y: int) -> int:
            return x + y


        #该函数的类型为Callable[[float, float], float]。
        def f3(x: float, y: float) -> float:
            return x * y


        #该函数的类型为Callable[..., str]。
        def f4(n: int, s: str, sep: str = "") -> str:
            return (n * (s + sep)).rstrip(sep)


        #该函数的类型为Callable[..., list]。
        def f5(*args: Any, reverse: bool = False) -> list:
            l = [s for s in args]
            if reverse:
                l.reverse()
                return l
            else:
                return l


        print(verbose(f1))
        print("")
        print(verbose(f2, 12, 34))
        print("")
        print(verbose(f3, 3.6, -1.7))    #报错。
        print("")
        print(verbose(f4, 3, "abc", sep="@"))    #报错。
        print("")
        print(verbose(f5, False, None, 0, reverse=True))    #报错。
```

请将上述代码保存为typing15.py，先通过如下命令行验证：

```
    $ python3 typing15.py
    executing f1()
    True

    executing f2(12, 34)
    46

    executing f3(3.6, -1.7)
    -6.12
```

```
    executing f4(3, abc, sep=@)
    abc@abc@abc

    executing f5(False, None, 0, reverse=True)
    [0, None, False]
```

再通过如下命令行验证：

```
    $ mypy typing15.py
    typing15.py:52: error: Argument 1 to "verbose" has incompatible type
"Callable[[float, float], float]"; expected "Callable[..., int]"  [arg-type]
    typing15.py:54: error: Argument 1 to "verbose" has incompatible type
"Callable[[int, str, str], str]"; expected "Callable[..., int]"  [arg-type]
    typing15.py:56: error: Argument 1 to "verbose" has incompatible type
"Callable[[VarArg(Any), DefaultNamedArg(bool, 'reverse')], List[Any]]"; expected
"Callable[..., int]"  [arg-type]
    Found 3 errors in 1 file (checked 1 source file)
```

仅使用Callable结构无法表示装饰器的类型，这是因为装饰器以一个可调用对象为参数，然后返回该可调用对象被装饰后的版本，所以装饰器的参数的类型和返回对象的类型是有相关性的，而单纯使用Callable结构无法反映这点——相对于表示参数类型的Callable结构，表示返回对象类型的Callable结构可能减少了原有参数，或增加了额外参数，因此无法基于位置保持参数的一一对应。

请注意在泛型中出现在函数标注不同位置的同一类型变量事实上起到了关联这些参数和返回值的作用：它使得这些参数和返回值必须是相同类型。基于这一灵感，PEP 612为typing模块增加了ParamSpec类和Concatenate结构。

ParamSpec对象被称为“参数规格变量（parameter specification variables）”，其实例化语法为：

```
    class typing.ParamSpec(name, bound=None[, {covariant=True|
contravariant=True}]
```

该语法与通过TypeVar创建类型变量的第二种语法一致，然而这一设计目前还只是概念阶段，其关键字bound、covariant和contravariant的语义尚未确定，因此在实际中创建参数规格变量时只能给出name参数。

一个ParamSpec对象就可以捕获任意多个可以基于位置对应的形式参数，以及所有仅关键字形式参数。但要注意，如果用（*args, **kwargs）表示被装饰的可调用对象的形式参数列表，那么ParamSpec对象将捕获对应*args的后面若干个形式参数，以及对应**kwargs的所有形式参数。每个ParamSpec对象都具有下列两个属性：

➤ args：引用一个ParamSpecArgs对象，代表被捕获的可以基于位置对应的形式参数形成的元组。
➤ kwargs：引用一个ParamSpecKwargs对象，代表被捕获的仅关键字形式参数形成的字典。

Concatenate结构的语法为：

Concatenate[*atype1*, ..., *atypeN*, *ParamSpecVariable*]

也就是在N个类型后面跟一个ParamSpec对象。每个Concatenate结构都代表一个完整的形式参数列表，其中前N个可以基于位置对应的形式参数的类型必须是atype1～atypeN，后续的所有形式参数（包括仅关键字形式参数）则被ParamSpec对象捕获。

当表示一个装饰器时，如果被装饰的可调用对象和被返回的可调用对象的形式参数列表完全相同，则可以直接使用同一个ParamSpec对象来表示整个形式参数列表，进而使两者的形式参数列表的类型相同。如果两者的形式参数列表不完全相同，则需用两个Concatenate结构来分别表示它们的类型：若Concatenate[t_1, ..., t_N, P]为被装饰的可调用对象的形式参数列表的类型，Concatenate[t'_1, ..., t'_M, P]为被返回的可调用对象的形式参数列表的类型，则意味着t_1~t_N是被移除的形式参数，t'_1~t'_M是被添加的形式参数，P则是保持不变的形式参数。下面的例子是后一种情况：

```
from typing import Any, Callable, ParamSpec, Concatenate
from functools import wraps

#定义一个参数规格变量。
P = ParamSpec('P')


#该装饰器将f(x, y, *, error)装饰为f(pos, neg, zero, y, *, error)。
def decorator(wrapped: Callable[Concatenate[Any, P], int])\
        -> Callable[Concatenate[str, str, str, P], str]:
    #这里用到了ParamSpec对象的args属性和kwargs属性。
    @wraps(wrapped)
    def wrapper(pos: str, neg: str, zero: str, *args: P.args, **kwargs:
P.kwargs) -> str:
        try:
            match wrapped(0, *args, **kwargs):
                case 1:
                    return f"{args[0]} is {pos}."
                case -1:
                    return f"{args[0]} is {neg}."
                case _:
                    return f"{args[0]} is {zero}."
        except Exception as e:
            if e == kwargs["error"]:
                return e.args[0]
            else:
                raise
    return wrapper
```

```
    #定义compare(x, y, *, error)，然后将其装饰为compare(pos, neg, zero, y, *,
    # error)。
    @decorator
    def compare(x: Any, y: Any, *, error: Exception = RuntimeError()) -> int:
        try:
            if y > x:
                return 1
            elif y < x:
                return -1
            else:
                return 0
        except Exception:
            raise error


    if __name__ == "__main__":
        print(compare('positive', 'negative', 'zero', 83.2,
error=Exception("Not comparable!")))
        print(compare('positive', 'negative', 'zero', -19, error=Exception("Not
comparable!")))
        print(compare('positive', 'negative', 'zero', 0, error=Exception("Not
comparable!")))
        print(compare('positive', 'negative', 'zero', 0j, error=Exception("Not
comparable!")))
        print(compare('+', '-', '0', 18, error=Exception("Not comparable!")))
        print(compare('+', '-', '0', -23.9, error=Exception("Not
comparable!")))
        print(compare('+', '-', '0', -0.0, error=Exception("Not comparable!")))
        print(compare('+', '-', '0', 1+9j, error=Exception("Not comparable!")))
```

请将上述代码保存为typing16.py，先通过如下命令行验证：

```
    $ python3 typing16.py
    83.2 is positive.
    -19 is negative.
    0 is zero.
    Not comparable!
    18 is +.
    -23.9 is -.
    -0.0 is 0.
    Not comparable!
```

然后通过如下命令行验证：

```
    $ mypy typing16.py
    Success: no issues found in 1 source file
```

15-8. 类型理论——其他类型操作

（标准库：内置类型、types、typing）

typing模块提供了Union结构，使我们可以将多个（完全不相关的）类型合并为一个联合类型，其语法为：

typing.Union[*type1*, *type2*, ..., *typeN*]

Union结构要求type1～typeN必须具有__or__和__ror__魔术属性。表3-6说明内置元类type具有__or__和__ror__。如果你自定义了某个类，而该类的元类不是type的子类，那最好在该元类中实现表3-6列出的所有属性。

Union结构本身的类型是typing._UnionGenericAlias。从其名字就可以看出，该类型是typing._GenericAlias的子类，因此Union结构在逻辑上是一种特殊的泛型别名。下面的例子说明了这点：

```
$ python3

>>> import typing
>>> u = typing.Union[int, str]
>>> type(u)
<class 'typing._UnionGenericAlias'>
>>> typing._UnionGenericAlias.__mro__
(<class 'typing._UnionGenericAlias'>, <class 'typing._NotIterable'>, <class
'typing._GenericAlias'>, <class 'typing._BaseGenericAlias'>, <class
'typing._Final'>, <class 'object'>)
>>>
```

然而需要强调的是，Union结构与泛型别名存在两个重要的不同之处。第一个不同之处是，虽然Union结构也不是真正的类，但可以作为issubclass()和isinstance()的第二个参数。请看下面的例子：

```
$ python3

>>> import typing
>>> typing.Union[int, str]
typing.Union[int, str]
>>> u = typing.Union[int, str]
>>> issubclass(int, u)
True
>>> isinstance('abc', u)
True
>>>
```

第二个不同之处是，虽然Union结构具有__mro_entries__属性，但依然不能出现在类参数列表中作为基类。请继续上面的例子来验证这点：

```
>>> hasattr(u, "__mro_entries__")
True
>>> class C(u):
...     pass
...
Traceback (most recent call last):
```

```
        File "<stdin>", line 1, in <module>
        File "/Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/
typing.py", line 1513, in __mro_entries__
          raise TypeError(f"Cannot subclass {self!r}")
          ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
      TypeError: Cannot subclass typing.Union[int, str]
      >>>
```

从被抛出异常的回溯信息可以看出，这一特性是__mro_entries__属性本身实现的，它会检查自己的调用者是否是Union结构，如果是则抛出一个TypeError异常。

Union结构中可以包含泛型，且这样的Union结构可以作为issubclass()第二个参数。然而要注意，issubclass()的第一个参数可以是一个泛型，但不可以是一个泛型别名。请看下面的例子：

```
      $ python3

      >>> import typing
      >>> T = typing.TypeVar('T')
      >>> class G(typing.Generic[T]):
      ...     pass
      ...
      >>> U1 = typing.Union[G, bool]
      >>> issubclass(bool, U1)
      True
      >>> issubclass(G, U1)
      True
      >>> issubclass(G[int], U1)
      Traceback (most recent call last):
        File "<stdin>", line 1, in <module>
        File "/Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/
typing.py", line 1668, in __subclasscheck__
          if issubclass(cls, arg):
             ^^^^^^^^^^^^^^^^^^^
      TypeError: issubclass() arg 1 must be a class
      >>>
```

这样的Union结构也可以作为isinstance()第二个参数，而其第一个参数既可以是泛型的实例，也可以是泛型别名的实例。请继续上面的例子以验证：

```
      >>> obj0 = G()
      >>> obj1 = G[int]()
      >>> obj2 = G[str]()
      >>> isinstance(True, U1)
      True
      >>> isinstance(False, U1)
      True
      >>> isinstance(None, U1)
      False
      >>> isinstance(obj0, U1)
      True
      >>> isinstance(obj1, U1)
      True
      >>> isinstance(obj2, U1)
      True
```

```
>>>
```

此外，包含泛型的Union结构自身的类型依然是typing._UnionGenericAlias，因此它并不是一个泛型，请继续上面的例子以验证：

```
>>> type(U1)
<class 'typing._UnionGenericAlias'>
>>>
```

Union结构中也可以包含泛型别名，但这样的Union结构不可以作为issubclass()或isinstance()的第二个参数。请继续上面的例子以验证：

```
>>> U2 = typing.Union[G[int], bool]
>>> issubclass(bool, U2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/
typing.py", line 1668, in __subclasscheck__
    if issubclass(cls, arg):
       ^^^^^^^^^^^^^^^^^^^
  File "/Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/
typing.py", line 1250, in __subclasscheck__
    raise TypeError("Subscripted generics cannot be used with"
    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
TypeError: Subscripted generics cannot be used with class and instance
checks
>>> isinstance(obj1, U2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/
typing.py", line 1664, in __instancecheck__
    return self.__subclasscheck__(type(obj))
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "/Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/
typing.py", line 1668, in __subclasscheck__
    if issubclass(cls, arg):
       ^^^^^^^^^^^^^^^^^^^
  File "/Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/
typing.py", line 1250, in __subclasscheck__
    raise TypeError("Subscripted generics cannot be used with"
    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
TypeError: Subscripted generics cannot be used with class and instance
checks
>>>
```

types模块提供了Union结构的简化写法：

```
type1 | type2 | ... | typeN
```

需要强调的是，通过这种方法获得的Union结构的类型是types.UnionType。下面的例子说明types.UnionType和泛型别名没有联系，也不具有__mro_entries__属性：

```
$ python3

>>> import types
>>> type(int | str)
<class 'types.UnionType'>
>>> types.UnionType.__mro__
(<class 'types.UnionType'>, <class 'object'>)
>>> hasattr(types.UnionType, "__mro_entries__")
False
>>>
```

然而在涉及Union结构的地方，这两种Union结构的行为是完全一致的。下面的例子说明，当用types模块提供的方法获得包含泛型的Union结构时，该结构本身是types.UnionType类型的，但给予指派后会自动变成typing._UnionGenericAlias类型：

```
$ python3

>>> import typing
>>> T = typing.TypeVar('T')
>>> class G(typing.Generic[T]):
...     pass
...
>>> U = G | bool
>>> type(U)
<class 'typing._UnionGenericAlias'>
>>> u = U[int]
>>> u
typing.Union[__main__.G[int], bool]
>>> type(u)
<class 'typing._UnionGenericAlias'>
>>>
```

在被应用时，Union结构和类型变量存在重叠的地方：我们也可以通过类型变量来限制函数的参数和返回值在指定的几个类型中选择。两者的区别在于，当多个参数和返回值的类型注解是同一类型变量时，意味着它们的类型必须一致；而使用Union结构则不存在这一限制。下面的例子说明了这点：

```
from typing import Union, TypeVar

T = TypeVar('T', int, str, bytes)


#该函数的参数x和y，以及返回值的可能类型都是int、str和bytes，然而三者的类型并不需要是
# 一致的，而可以存在多种组合。   这导致无法用一个类型变量T来作为它们的类型注解，只能使
# 用Union结构。
#def add_concate(x: T, y: T) -> T:
def add_concate(x: Union[int, str, bytes], y: Union[int, str, bytes]) ->
Union[int, str, bytes]:
    try:
        if isinstance(x, int):
            if isinstance(y, int):
                return x + y
            elif isinstance(y, str):
```

```
                    return x * y
                elif isinstance(y, bytes) or isinstance(y, bytearray):
                    return x * y
                else:
                    raise TypeError("y is of bad type!")
            elif isinstance(x, str):
                if isinstance(y, int):
                    return x * y
                elif isinstance(y, str):
                    return x + y
                elif isinstance(y, bytes) or isinstance(y, bytearray):
                    raise TypeError("x's type and y's type are inconsistent!")
                else:
                    raise TypeError("y is of bad type!")
            elif isinstance(x, bytes) or isinstance(x, bytearray):
                if isinstance(y, int):
                    return x * y
                elif isinstance(y, str):
                    raise TypeError("x's type and y's type are inconsistent!")
                elif isinstance(y, bytes) or isinstance(y, bytearray):
                    return x + y
                else:
                    raise TypeError("y is of bad type!")
            else:
                raise TypeError("x is of bad type!")
        except TypeError as e:
            return e.args[0]


    if __name__ == "__main__":
        print(add_concate(2, 3))
        #使用T时报错。
        print(add_concate(2, 'abc'))
        #使用T时报错。
        print(add_concate(2, b'abc'))
        #使用T时报错。
        print(add_concate(2, bytearray(b'abc')))
        #总是报错。
        print(add_concate(2, 3.0))
        print("")
        #使用T时报错。
        print(add_concate('xyz', 3))
        print(add_concate('xyz', 'abc'))
        #使用T时报错。
        print(add_concate('xyz', b'abc'))
        #使用T时报错。
        print(add_concate('xyz', bytearray(b'abc')))
        #总是报错。
        print(add_concate('xyz', 3.0))
```

请将上述代码保存为typing17.py，先通过如下命令行验证其功能：

```
$ python3 typing17.py
5
abcabc
b'abcabc'
bytearray(b'abcabc')
y is of bad type!

xyzxyzxyz
xyzabc
```

```
      x's type and y's type are inconsistent!
      x's type and y's type are inconsistent!
      y is of bad type!
```

接下来通过如下命令行验证当使用Union结构时仅两处报错：

```
      $ mypy typing17.py
      typing17.py:54: error: Argument 2 to "add_concate" has incompatible type
"float"; expected "Union[int, str, bytes]"  [arg-type]
      typing17.py:64: error: Argument 2 to "add_concate" has incompatible type
"float"; expected "Union[int, str, bytes]"  [arg-type]
      Found 2 errors in 1 file (checked 1 source file)
```

最后将三处“Union[int, str, bytes]”都替换成“T”，通过如下命令行验证当使用类型变量时将有8处报错：

```
      $ mypy typing17.py
      typing17.py:48: error: Value of type variable "T" of "add_concate" cannot
be "object"  [type-var]
      typing17.py:50: error: Value of type variable "T" of "add_concate" cannot
be "object"  [type-var]
      typing17.py:52: error: Value of type variable "T" of "add_concate" cannot
be "object"  [type-var]
      typing17.py:54: error: Value of type variable "T" of "add_concate" cannot
be "float"  [type-var]
      typing17.py:57: error: Value of type variable "T" of "add_concate" cannot
be "object"  [type-var]
      typing17.py:60: error: Value of type variable "T" of "add_concate" cannot
be "Sequence[object]"  [type-var]
      typing17.py:62: error: Value of type variable "T" of "add_concate" cannot
be "Sequence[object]"  [type-var]
      typing17.py:64: error: Value of type variable "T" of "add_concate" cannot
be "object"  [type-var]
      Found 8 errors in 1 file (checked 1 source file)
```

Union结构还具有如下性质：

➤ 1. 嵌套会被自动展开：

```
      typing.Union[typing.Union[int, str], float] == typing.Union[int, str,
float]
      (int | str) | float == int | str | float
```

➤ 2. 冗余的类型会被自动删除：

```
typing.Union[int, str, int] == typing.Union[int, str]
int | str | int == int | str
```

➤ 3. 类型的顺序不重要：

```
typing.Union[int, str] == typing.Union[str, int]
int | str == str | int
```

最后，typing模块还提供了Optional结构，其语法为：

```
typing.Optional[type]
```

它其实就等价于typing.Union[type, None]。下面的例子说明了这点：

```
$ python3

>>> import typing
>>> U1 = typing.Optional[int]
>>> type(U1)
<class 'typing._UnionGenericAlias'>
>>> U2 = typing.Union[int, None]
>>> U1 == U2
True
>>>
```

在第5章，我们讨论过通过反射机制利用变量实现常量的技巧，然而它是比较麻烦的。很多时候，我们只想确保脚本中没有因误操作而改变被当成常量的变量所引用的对象，此时通过结合使用Final结构和静态类型检查工具达到这一目的更加简便。Final结构的语法为：

```
typing.Final
typing.Final[type]
```

以上面的Final结构作为一个变量的类型注解时，仅表明该变量引用的对象不能改变，而不限制被引用对象的类型。而以下面的Final结构作为一个变量的类型注解时，还会额外限制被应用对象的类型是type。显然，Final结构只能出现在变量标注中，而不能出现在函数标注中。下面是一个例子：

```
import typing

NAME: typing.Final = "abc"

NAME = "def"    #报错。

class C():
    NUM: typing.Final[int] = 3.0    #报错。

C.NUM = 3    #报错。
```

请将上述代码保存为typing18.pyi，然后通过如下命令行验证：

```
$ mypy typing18.pyi
typing18.pyi:5: error: Cannot assign to final name "NAME"  [misc]
typing18.pyi:8: error: Incompatible types in assignment (expression has
type "float", variable has type "int")  [assignment]
typing18.pyi:10: error: Cannot assign to final attribute "NUM"  [misc]
Found 3 errors in 1 file (checked 1 source file)
```

需要说明的是，typing.Final本身并不是泛型，但当给它指派了一个类型后，将得到一个泛型别名。请通过如下命令行和语句验证：

```
$ python3

>>> import typing
>>> type(typing.Final)
<class 'typing._SpecialForm'>
>>> type(typing.Final[int])
<class 'typing._GenericAlias'>
>>>
```

本章前面已经给出了通过__init_subclass__魔术属性使得一个类不能作为基类的例子。我们也可以结合使用@typing.final装饰器和静态类型检查工具达到相同目的。此外，当用@typing.final来装饰一个方法时，会使得该方法不能被子类重写。下面的例子说明了该装饰器的用法：

```
import typing


class A:
    #该方法不能被重写。
    @typing.final
    def a(self):
        pass

    def b(self):
        pass


#该类不能作为其他类的基类。
```

```
    @typing.final
    class B(A):
        def a(self) -> int:    #报错。
            return 0

        def b(self) -> int:
            return 0


    class C(B):    #报错。
        pass
```

请将上述代码保存为typing19.py，然后通过如下命令行验证：

```
    $ mypy typing19.py
    typing19.py:17: error: Cannot override final attribute "a" (previously
declared in base class "A")  [misc]
    typing19.py:24: error: Cannot inherit from final class "B"  [misc]
    Found 2 errors in 1 file (checked 1 source file)
```

Python 3.11给@typing.final增加了这样一个功能：被其装饰的对象会被添加__final__属性，并使该属性引用True。这意味着我们可以通过getattr(obj, "__final__", False)来判断一个对象是否有被@typing.final装饰，进而实现动态类型检查。请继续上面的例子，通过如下命令行和语句验证：

```
    $ python3 -i typing19.py

    >>> A.a.__final__
    True
    >>> B.__final__
    True
    >>>
```

---

typing模块提供的TypeGuard结构与Final结构类似，本身不是泛型，但可以被指派一个类型，并得到一个泛型别名。下面的命令行和语句验证了这点：

```
    $ python3

    >>> import typing
    >>> type(typing.TypeGuard)
    <class 'typing._SpecialForm'>
    >>> type(typing.TypeGuard[int])
    <class 'typing._GenericAlias'>
    >>>
```

但与Final结构不同，TypeGuard结构必须被指派后才能使用，且总是作为函数返回值的类型注解，因此其语法为：

typing.TypeGuard[*type*]

TypeGuard结构只用于这样一种函数：该函数只有一个形式参数，而该参数的类型注解指定了一个类型范围，而该函数会根据通过该形式参数传入的对象的类型返回一个布尔值，进而把其类型注解指定的类型范围划分为两个子范围。而TypeGuard结构被指派的类型代表的是使该函数返回True的子范围。请看下面的例子：

```
from typing import TypeGuard, Tuple


#该TypeGuard结构表明，当is_record()返回True时，通过data参数传入的是一个(str, int)
# 格式的二元组。
def is_record(data: tuple) -> TypeGuard[Tuple[str, int]]:
    if len(data) != 2:
        return False
    if isinstance(data[0], str) and isinstance(data[1], int):
        return True
    else:
        return False


print(is_record((0, 1)))    #显示False。
print(is_record(('abc', 0, 1)))    #显示False。
print(is_record(('abc', 0)))    #显示True。
print(is_record((b'abc', 0.0)))    #显示False。
```

请将上述代码保存为typing20.py，先通过如下命令行验证其功能：

```
$ python3 typing20.py
False
False
True
False
```

然后再通过如下命令行进行静态类型检查：

```
$ mypy typing20.py
Success: no issues found in 1 source file
```

---

typing模块提供了两种机制来为已有类型创建别名。第一种机制就是简单的赋值语句，例如：

```
new_type = type
```

其中type是一个已经存在的类型，而该赋值语句会使new_type引用同一个类型。为了强调该赋值语句创建了一个类型别名，我们可以将typing.TypeAlias作为new_type的类型注解，即：

```
new_type: typing.TypeAlias = type
```

下面的例子说明了这种创建类型别名的方法：

```
$ python3

>>> import typing
>>> type(typing.TypeAlias)
<class 'typing._SpecialForm'>
>>> INT: typing.TypeAlias = int
>>> INT is int
True
>>> INT(34.5)
34
>>>
```

第二种机制是使用类typing.NewType，其实例化语法为：

```
class typing.NewType(name, type)
```

该类的实例是一个可调用对象，必须赋值给名为name的标识符，调用时的语法为：

```
name(obj)
```

其作用是将通过obj传入的对象原封不动的返回。通过这种方式创建的类型别名name将被视为type的子类型，而将一个类型为type的对象作为name的参数后，得到的对象在静态类型检查时将被视为name类型而非type类型，而对Python脚本的执行毫无影响。需要强调的是，由于NewType对象并不是类，所以不能通过将其作为基类的方式创建它的子类型。但一个NewType对象可以作为typing.NewType的type参数，这样创建的新NewType对象将被视为原NewType对象的子类型。下面是一个例子：

```
from typing import NewType, Tuple

#创建Tuple[str, int]的一个子类型，并将其命名为Record。
Record = NewType('Record', Tuple[str, int])
```

```
#创建Record的一个子类型，并将其命名为Item。
Item = NewType('Item', Record)


#该函数需被传入一个Record类型或其子类型的对象。
def show_record(data: Record):
    print(data)


#t1是Tuple[str, int]类型。
t1 = ('abc', 0)
#t2是Record类型。
t2 = Record(t1)
#t3是Item类型。
t3 = Item(t2)
#显示True，说明t2和t1引用同一个对象。
print(t2 is t1)
#显示True，说明t3和t2引用同一个对象。
print(t3 is t2)

#传入Tuple[str, int]类型对象，报错。
show_record(t1)
#传入Record类型对象，不报错。
show_record(t2)
#传入Record的子类型对象，不报错。
show_record(t3)
```

请将上述代码保存为typing21.py，先通过如下命令行验证其功能：

```
$ python3 typing21.py
True
True
('abc', 0)
('abc', 0)
('abc', 0)
```

然后再通过如下命令行进行静态类型检查：

```
$ mypy typing21.py
typing21.py:27: error: Argument 1 to "show_record" has incompatible type
"Tuple[str, int]"; expected "Record"  [arg-type]
Found 1 error in 1 file (checked 1 source file)
```

最后，Python 3.11为typing模块增加了三个配合静态类型检查工具使用的辅助函数。

typing.assert_type()的语法为：

```
typing.assert_type(obj, type)
```

在进行静态类型检查时，如果obj的类型不是type或type的子类型，则会报错。而在运行时，该函数会原封不动地返回obj。

typing.assert_nerver()的语法为：

```
typing.assert_never(arg: Never) -> Never
```

在进行静态类型检查时，如果typing.assert_nerver()会被执行，则会报错。而在运行时，如果typing.assert_nerver()被执行，则会抛出AssertionError异常（将在第16章讨论）。

typing.reveal_type()的语法为：

```
typing.reveal_type(obj)
```

在进行静态类型检查时，会以“Revealed type is xxx”的形式报告obj的类型。而在运行时，该函数会以“Runtime type is xxx”的形式报告obj的类型，然后原封不动地返回obj。

下面的例子说明了上述三个函数的用法：

```
import typing
from numbers import Number


def str_or_num(obj: str | int | float):
    print(typing.assert_type(obj, str | complex))    #报错。
    match obj:
        case str():
            #先报告obj的类型，再显示obj。
            print(typing.reveal_type(obj))
        case int():
            #先报告obj的类型，再显示obj。
            print(typing.reveal_type(obj))
        case _ as unreachable:
            try:
                typing.assert_never(unreachable)    #报错。
            except AssertionError:
                #先报告obj的类型，再显示obj。
                print(typing.reveal_type(obj))
    print("")


str_or_num("abc")
str_or_num(b"abc")    #报错。
str_or_num(0)
str_or_num(True)
str_or_num(1.3)
str_or_num(None)    #报错。
```

请将上述代码保存为typing22.py，先通过如下命令行验证三个函数在运行时的功能：

```
$ python3 typing22.py
abc
Runtime type is 'str'
abc

b'abc'
Runtime type is 'bytes'
b'abc'

0
Runtime type is 'int'
0

True
Runtime type is 'bool'
True

1.3
Runtime type is 'float'
1.3

None
Runtime type is 'NoneType'
None
```

再通过如下命令行验证三个函数在进行静态类型检查时的功能：

```
$ mypy typing22.py
typing22.py:6: error: Expression is of type "Union[str, int, float]", not
"Union[str, complex]"  [assert-type]
typing22.py:10: note: Revealed type is "builtins.str"
typing22.py:13: note: Revealed type is "builtins.int"
typing22.py:16: error: Argument 1 to "assert_never" has incompatible type
"float"; expected "NoReturn"  [arg-type]
typing22.py:19: note: Revealed type is "builtins.float"
typing22.py:24: error: Argument 1 to "str_or_num" has incompatible type
"bytes"; expected "Union[str, int, float]"  [arg-type]
typing22.py:28: error: Argument 1 to "str_or_num" has incompatible type
"None"; expected "Union[str, int, float]"  [arg-type]
Found 4 errors in 1 file (checked 1 source file)
```

值得一提的是，typing模块还提供了@typing.dataclass_transform装饰器。该装饰器被用来装饰另一个装饰器，以使后者能被视为一个数据类。标准库中的dataclasses模块实现了数据类，本书不详细讨论。

15-9. 多态

（标准库：functools、typing）

至此我们已经讨论完了Python的类型理论。下面让我们讨论在第5章已经提到过的多态。事实上，多态可以被看成类型理论的延伸，其本质特征是同一个接口（函数或方法）因实际参数类型不同而做出不同的行为，但这些不同的行为的内在逻辑保持不变。在这个前提下，

计算机科学中的多态存在三种形式：“特定多态（ad hoc polymorphism）”、“参数多态（parametric polymorphism）”和“子类型多态（subtyping polymorphism）”。下面依次讨论。

特定多态指的是为每种参数类型的组合提供一个接口的实现，而该接口会根据实际参数类型的组合自动选择使用哪个实现。同一接口的多个实现相互间在保持内在逻辑不变的前提下，具体执行步骤可以差别很大。

对于静态类型的编程语言来说，实现特定多态相对容易：只需要定义一组函数名/方法名相同、形式参数的类型组合不同的函数或/方法就可以了。由于编译器并非仅根据函数名/方法名来识别函数/方法，而是要结合参数列表，所以每个这样的函数/方法对于编译器来说都是独立的函数/方法，编译器甚至不会意识到这是一种多态的表现形式。

对于动态类型的编程语言来说，由于变量的类型要到运行时才能确定，所以只能将该接口的所有实现都写在一个包含分支语句的函数体中，并通过动态判断变量类型的机制——例如issubclass()和isinstance()——决定执行哪个分支。而为了支持静态类型检查，typing模块提供了@typing.overload装饰器，下面的例子说明了其用法：

```
import typing
from numbers import Number
from collections.abc import ByteString

U = typing.Union[int, float, complex]

#该函数实现了特定多态。
@typing.overload    #注册add()的第一个实现。
def add(x: U, y: U) -> U: ...
@typing.overload    #注册add()的第二个实现。
def add(x: str, y: str) -> str: ...
@typing.overload    #注册add()的第三个实现。
def add(x: ByteString, y: ByteString) -> ByteString: ...
def add(x, y):    #定义add()。
    #add()的第一个实现。
    if isinstance(x, Number) and isinstance(y, Number):
        return x + y
    #add()的第二个实现。
    elif isinstance(x, str) and isinstance(y, str):
        return "".join([x, y])
    #add()的第三个实现。
    elif isinstance(x, ByteString) and isinstance(y, ByteString):
        if isinstance(x, bytes):
            return b"".join([x, y])
        else:
            return bytearray(b"").join([x, y])
    #当实际参数的类型组合不匹配任何一个实现时，抛出TypeError异常。
    else:
        raise TypeError


if __name__ == "__main__":
    print(add(0, 15))
    print(add(8j, 6-9j))
```

```
print(add('a', 'b'))
print(add(b'a', b'b'))
print(add(b'a', bytearray(b'b')))
print(add(bytearray(b'a'), bytearray(b'b')))
print(add('a', b'b'))
```

请将上述代码保存为polymorphism1.py，先通过如下命令行验证：

```
$ python3 -i polymorphism1.py
15
(6-1j)
ab
b'ab'
b'ab'
bytearray(b'ab')
Traceback (most recent call last):
  File "/Users/wwy/polymorphism1.py", line 40, in <module>
    print(add('a', b'b'))
          ^^^^^^^^^^^^^^
  File "/Users/wwy/polymorphism1.py", line 30, in add
    raise TypeError
    ^^^^^^^^^^^^^^^
TypeError
```

再通过如下命令行验证：

```
$ mypy polymorphism1.py
polymorphism1.py:40: error: No overload variant of "add" matches argument
types "str", "bytes"  [call-overload]
polymorphism1.py:40: note: Possible overload variants:
polymorphism1.py:40: note:     def add(x: Union[int, float, complex], y:
Union[int, float, complex]) -> Union[int, float, complex]
polymorphism1.py:40: note:     def add(x: str, y: str) -> str
polymorphism1.py:40: note:     def add(x: ByteString, y: ByteString) ->
ByteString
Found 1 error in 1 file (checked 1 source file)
```

从上面的例子可以看出，在Python中实现特定多态本质上是通过在函数体内进行类型判断实现的，相应函数或方法就是接口。@typing.overload装饰器的作用，则是将多个实现注册到该接口：每个实现都通过函数标注来描述，且需要被该装饰器装饰。需要强调的是，如果不通过@typing.overload装饰器将实现注册到接口，那么静态类型检测工具无法识别出该函数或方法是一个特定多态，但这对Python脚本的执行不会造成任何不良影响。

@typing.overload装饰器还反映出这样一个事实：在Python中特定多态是通过“重载（overloads）”实现的。上面的例子被称为“函数重载（function overloading）”。而当我们重写__eq__、__add__和__contains__之类的与运算符相关的魔术属性时，则被称为“运算符重载（operator overloading）”。注意上面的例子中有意通过字符串/字节串的join()属性完成了字符串/字节串的拼接，而这其实完全可以通过+实现，也就是说add()也可以改写为：

```
@typing.overload
def add(x: U, y: U) -> U: ...
@typing.overload
def add(x: str, y: str) -> str: ...
@typing.overload
def add(x: ByteString, y: ByteString) -> ByteString: ...
def add(x, y):
    return x + y
```

而这里用到的事实就是Python解释器内置的+运算符其实已经进行了运算符重载。

Python 3.11给typing模块增加了两个函数，使得通过@typing.overload装饰器注册的重载可以在运行时被操作。第一个函数是typing.getoverloads()，其语法为：

typing.get_overloads(*func*)

其功能是以函数对象列表的形式取得接口func重载的所有实现；如果func没有重载任何实现（即并非特定多态），则返回一个空列表。第二个函数是typing.clear_overloads()，其语法为：

typing.clear_overloads()

这会清空之前所有接口重载的所有实现。请通过如下命令行和语句验证：

```
$ python3

>>> from polymorphism1 import *
>>> typing.get_overloads(add)
[<function add at 0x10a4a8360>, <function add at 0x10a47b880>, <function
add at 0x10a47b920>]
>>> typing.clear_overloads()
>>> typing.get_overloads(add)
[]
>>>
```

参数多态指的是该接口的参数以及提供该接口的对象的类型是泛型，而在访问该接口时确定具体执行的操作被称为“分派（dispatch）”。

分派被分为两种：

➤ “单分派（single dispatch）”：仅根据实际参数的类型确定具体执行的操作。
➤ “多分派（multiple dispatch）”：同时根据提供该接口的对象的类型和实际参数的类型确定具体执行的操作。

对于静态类型的编程语言来说，参数多态往往表现为所谓的“模板（templates）”：模板本质上是包含若干泛型的函数，即所谓的“泛型函数（generic functions）”。由于泛型本身可以在声明语句中使用，并在编译时被替代为具体的类型，所以可以通过一个泛型函数实现针对泛型所覆盖的所有类型的操作。当然，这些操作不仅要保持内在逻辑不变，具体执行步骤也是相同的，不同之处仅存在于这些类型实现同一操作的同名属性的函数体内。反过来说，用模板实现的参数多态必然较为死板。此外，静态类型的编程语言既可以实现单分派，也可以实现多分派，取决于该语言的设计者。

对于动态类型的编程语言来说，参数多态同样涉及泛型函数，然而它们没有声明语句（或者声明语句对解释执行的结果没有影响），所以通常让该泛型函数实现最基本的实现，然后再针对不同的实际参数的类型组合注册不同的实现。这些实现每个都对应一个函数体，通过分派机制，每次都只有其中一个函数体被执行。此外，动态类型的编程语言只能实现单分派。functools模块提供了@singledispatch装饰器来将一个函数包装为一个泛型函数，并为其提供了用于注册实现的register()属性。而该模块提供的@singledispatchmethod()装饰器则是针对方法的，功能与前者相同。下面的例子说明了@functools.singledispatch的用法：

```
import typing
from collections.abc import Mapping, Sequence
from functools import singledispatch

U = typing.Union[str, bytes, bytearray]


#定义一个泛型函数。   参数obj的类型是泛型，决定分派的结果。   参数maxrow的类型是int，
# 与分派无关。
@singledispatch
def table_gen(obj, maxrow=5):
    return [(1, obj)]


#以第一种方式将针对Sequence的实现注册给泛型函数。
@table_gen.register
def _(obj: Sequence, maxrow=5):
    table = list()
    row = 1
    for ele in obj:
        table.append((row, ele))
        if row == maxrow:
            break
        row += 1
    return table


#以第二种方式将针对Mapping的实现注册给泛型函数。
@table_gen.register(Mapping)
def _(obj, maxrow=5):
    table = list()
    row = 1
    for k, v in obj.items():
```

```
            table.append((row, k, v))
            if row == maxrow:
                break
            row += 1
        return table


#以第三种方式将针对字符串和字节串的实例注册给泛型函数。
table_gen.register(U, lambda obj, maxrow: [(1, obj)])


if __name__ == "__main__":
    #调用泛型函数本身。
    print(table_gen(0j))
    #调用针对字符串和字节串的实现。
    print(table_gen('abc', 5))
    #调用针对Sequence的实现。
    print(table_gen([1.6, -8.2, 7.9, 5.4, -0.9], 3))
    #调用针对Mapping的实现。
    print(table_gen({'a': b'a', 'b': b'b'}))
    #调用泛型函数本身。
    print(table_gen({'x', 'y', 'z'}))
```

请将上述代码保存为polymporphism2.py，先通过如下命令行验证：

```
$ python3 polymorphism2.py
[(1, 0j)]
[(1, 'abc')]
[(1, 1.6), (2, -8.2), (3, 7.9)]
[(1, 'a', b'a'), (2, 'b', b'b')]
[(1, {'y', 'z', 'x'})]
```

再通过如下命令行验证：

```
$ mypy polymorphism2.py
polymorphism2.py:42: error: No overload variant of "register" of
"_SingleDispatchCallable" matches argument types "object", "Callable[[Any, Any],
List[Tuple[int, Any]]]"  [call-overload]
polymorphism2.py:42: note: Possible overload variants:
polymorphism2.py:42: note:     def register(self, cls: Type[Any], func:
None = ...) -> Callable[[Callable[..., Any]], Callable[..., Any]]
polymorphism2.py:42: note:     def register(self, cls: Callable[..., Any],
func: None = ...) -> Callable[..., Any]
polymorphism2.py:42: note:     def register(self, cls: Type[Any], func:
Callable[..., Any]) -> Callable[..., Any]
Found 1 error in 1 file (checked 1 source file)
```

现在让我们结合上面的例子来阐述@functools.singledispatch的工作机制。被该装饰器装饰过的函数的第一个参数自动被视为泛型：如果它有类型注解，则该类型注解会被解读为对泛型的限制；否则，它的默认类型注解typing.Any会导致该泛型没有任何限制。该泛型函数的分派只考虑其第一个参数，与其他的参数无关，哪怕后者通过类型注解被声明为具有泛型类型。由于我们可以给第一个参数传入一个容器，所以分派时只考虑第一个参数不会使Python中的参数多态比其他动态类型编程语言的参数多态功能弱。

在定义了泛型函数后，默认的分派规则是不论第一个参数被传入什么类型的实际参数，都执行该泛型函数的函数体。要想使该分派有意义，就必须为该泛型函数注册实现：每个实现针对一种或一类类型，本质上是另一个函数。@functools.singledispatch会自动使得泛型函数具有register()属性，而通过该属性注册实现的方式有三种：

➤ 将register()当成实现的装饰器，通过实现第一个参数的类型注解来指定该实现所针对的类型。在上面的例子中，针对collections.abc.Sequence的实现是以这种方式注册的。
➤ 将register()当成装饰器生成函数，给其传入实现所针对的类型，并以返回值来装饰实现。在上面的例子中，针对collections.abc.Mapping的实现是以这种方式注册的。
➤ 将register()当成普通函数，其第一个参数被传入实现所针对的类型，第二个参数被传入实现本身。在上面的例子中，针对字符串和字节串的实例是以这种方式注册的。

注意针对字符串和字节串的实例同时说明可以针对Union结构注册实现。然而当通过mypy对脚本进行静态类型检查时，在这里报错，这是因为在本书被编写时的mypy版本（0.991）还不能识别针对Union结构注册的实现。

当一个泛型函数被注册了实现后，当其被调用时会按照如下步骤确定调用哪个实现：

➤ 步骤一：如果第一个实际参数的类型被直接注册，则执行相应实现。注意当一个Union结构被注册后，其内的每个类型都相当于被直接注册。
➤ 步骤二：如果第一个实际参数的类型没有被直接注册，则检查它是否是某个被注册的类型的子类型，如果是则执行相应实现。
➤ 步骤三：如果通过前两个步骤没能确定执行哪个实现，则执行泛型函数本身。

上述规则使的注册的类型相互间不需要完全是平行的，例如在上面的例子中str、bytes和bytearray都是collections.abc.Sequence的子类型，然而当第一个实际参数是字符串或字节串时，按照上面的规则通过步骤一就确定了执行针对字符串和字节串的实现，而不是执行针对collections.abc.Sequence的实现。

@functools.singledispatchmethod与@functools.singledispatch的区别只有一点：它装饰的是方法而非函数，所以分派时考虑的是第二个参数而非第一个（cls和self）。显然，@functools.singledispatchmethod只能装饰方法和类方法，静态方法则应被视为等同于函数，通过@functools.singledispatch来装饰。此外需要强调，为了保证register()有效，这两个装饰器都必须被作为最外层装饰器来使用。

子类型多态指的是一个类的方法能够以其子类的实例或子类本身为参数。注意子类可以重写该类的方法，也可以直接继承该类的方法，两种情况都不影响子类型多态的成立。

在Python中广泛存在子类型多态。可以认为，表3-5和表3-6中的每个属性都是子类型多态的一个例子。

举例来说，我们自定义类时很少重写__new__魔术属性，因此当实例化该类时事实上调用的是object实现的__new__，但给cls参数传入的类却不是object，而是自定义类，但__new__依然能正常工作。

另一个例子是print()可以显示任何对象，这本质上是调用该对象的__str__魔术属性。有些类型的对象会重写__str__，但没有重写__str__的对象会调用它所属类的基类实现的__str__，在后一种情况下，基类实现的__str__必须兼容子类的实例。

抽象基类的每个非抽象子类都需要重写前者的每个抽象方法，而这同样属于子类型多态。最典型的例子是numbers模块中的抽象基类，它们形成了如图15-2所示的“数字塔（numerical tower）”。位于该塔某一层的数字类型必须实现塔顶到该层的抽象基类中的所有抽象方法，但具体实现方式可以与同属于该层的其他数字类型不同。例如前面的例子abc2.py中的BiIntVector类与内置类型complex同属于数字塔中的“Complex”层，但它们对Complex包含的抽象方法的实现大相径庭，而这是典型的子类型多态。

15-10. 弱引用

（语言参考手册：3.3.2.4）
（标准库：内置类型、内置异常、weakref、sys）

根据第3章描述的数据模型，在Python中想访问一个对象就必须获得该对象的一个引用，然而这意味着该对象能够被访问的前提是其引用数不为0。如果一个对象的引用数为0，就意味着没有任何其他对象需要访问它了，因此在下次垃圾回收机制被启动时该对象就会被销毁，而如果该对象所属类具有__del__魔术属性，那么__del__会在被销毁时自动调用。这个模型看起来似乎是完美的，但事实上在一些特殊场景下存在微妙的问题。

让我们先来看这样一段代码：

```
s = 'abc'
del s
print(s)
```

如果将这段代码保存到一个Python脚本中，然后执行它，那么结果是抛出了NameError异常，因为在执行“print(s)”时标识符s已经被从__main__模块的变量字典中删除。这里需要注意的是，在执行“print(s)”时，标识符s和字符串“abc”其实已经进入了垃圾回收流程，因为“del s”使得s的引用数变为0，而“abc”只被s引用，s被销毁后它的引用数也会变为0。

然而我们在第11章讨论字符串驻留技术时已经提到了，Python脚本中出现的所有标识符和字符串都会被自动插入驻留字符串表。那么假如驻留字符串表直接引用这些标识符和字符串

会造成什么影响呢？上面这段代码的运行结果不会改变，但标识符s和字符串“abc”将一直驻留在内存中，直到脚本执行完成。这是因为驻留字符串表为它们提供了额外的引用数。

让我们再考虑另一个场景。本章前面已经说明，abc.ABCMeta的__subclasscheck__在执行__subclasshook__时，会非常智能地查找指定抽象基类有哪些派生抽象基类。然而第3章中的数据模型仅通过__bases__和__mro__记录了一个类的基类，无法实现快速查找一个类的子类。这个问题其实是这样解决的：每个类对象都有一个函数类属性__subclasses__()，调用它会返回其直接子类形成的列表。请看下面的例子：

```
    $ python3

    >>> int.__subclasses__()
    [<class 'bool'>, <enum 'IntEnum'>, <flag 'IntFlag'>, <class
're._constants._NamedIntConstant'>]
```

虽然__subclasses__()是一个函数，但为了保证其执行速度，每个类必须通过某种方式记录自己的所有直接子类。然而若通过引用记录这些直接子类，就会形成循环引用。假设A是B的直接基类，下面的例子说明仅通过del删除标识符A是无法让类A的引用数为0的，因为类B通过__mro__引用了类A：

```
    $ python3

    >>> class A:
    ...     pass
    ...
    >>> class B(A):
    ...     pass
    ...
    >>> B.__mro__
    (<class '__main__.B'>, <class '__main__.A'>, <class 'object'>)
    >>> del A
    >>> B.__mro__
    (<class '__main__.B'>, <class '__main__.A'>, <class 'object'>)
    >>> A
    Traceback (most recent call last):
      File "<stdin>", line 1, in <module>
    NameError: name 'A' is not defined
    >>>
```

如果再通过del删除标识符B，那么类B的引用数先变为0，而类B被销毁后类A的引用数也会变成0，使得两者都被销毁。但如果类A为了支持__subclasses__()而直接引用了B，那么即便通过del删除了标识符B，类B的引用数也不会变为0，因此类A的引用数也不会变为0，导致两者无法被销毁。这就是循环引用。类似的，任务和Future对象之间也存在产生循环引用的可能。

想要解决上面两个场景中的微妙问题，就必须有一种能够访问到某个对象，却不增加它的引用数的方法，而这就是所谓的“弱引用（weak reference）”。第11章已经提到了，驻留字符串表是通过弱引用来记录字符串的。第14章已经提到，Future对象通过弱引用来关联它所属于的任务。而__subclasses__()也是通过弱引用访问它所在类的直接子类的。标准库中

的weakref模块为我们提供了使用弱引用的接口，本章的最后一节将讨论weakref模块的使用。

---

weakref模块提供了ref类（注意该类名是小写的，事实上违反了PEP 8推荐）来创建弱引用，其实例化语法为：

```
class weakref.ref(obj[, callback])
```

其必须的obj参数必须是一个支持弱引用的对象，而可选的callback参数则用于注册当弱引用的对象被销毁时被调用的回调函数。

ref类成功实例化后得到的对象就代表着一个对obj的弱引用，而该弱引用是可调用的，调用结果遵守如下规则：如果弱引用被调用时，它引用的对象还存在，则返回该对象；否则返回None。我们把引用的对象还存在的弱引用称为“存活的（alive）”，而把引用的对象已经不存在的弱引用称为“死亡的（dead）”。weakref定义了ReferenceType常量来代表弱引用的类型，它其实就等于ref类。

并非所有Python对象都能被弱引用。能够被弱引用的对象包括如下9种：

- 任何类型对象，包括内置类型和自定义类。
- 用Python编写的函数。（但不能是用C编写的函数。）
- 自定义类的自定义方法。
- 具有__weakref__属性的自定义类的实例。
- 集合类型、deque和array的实例。
- 生成器。
- 文件对象和套接字对象。
- 正则表达式对象。
- 代码对象。

需要说明的是，自定义类在三种情况下具有__weakref__属性：

- 类定义语句中没有包含对__slots__的设置。
- 类定语语句中包含对__slots__的设置，但“__weakref__”在__slots__列出的标识符中。
- 类定义语句中包含对__slots__的设置，且没有列出__weakref__，但该类的某个基类具有__weakref__。

注意上面第3种情况说明，__weakref__与__dict__存在相似性，但它并不是特殊属性，可以被子类继承。如果一个自定义类具有__weakref__属性，则它会引用一个数据描述器，但我们不应手工设置它。内置类型都不具有__weakref__属性。

下面用一个例子说明如何创建和使用弱引用。为了简便，它只涉及上面列出的9种能够被弱引用的对象中的前4种：

```
import typing
import weakref


#定义一个函数。
def f1():
    print("I am a function, my name is 'f1'.")


#定义一个类，它没有设置__slots__，因此具有__weakref__属性。
class C1():
    def __init__(self, who):
        self.who = who

    def m1(self):
        print(f"I am '{self.who}'.")

    @classmethod
    def m2(cls):
        print(f"My class is '{cls.__name__}'.")


#定义一个类，它设置了__slots__，但由于以C1为基类，所以也具有__weakref__属性。
class C2(C1):
    __slots__ = 'who'

    @staticmethod
    def m3():
        print(f"I am 'C2'.")


#定义一个类，它不具有__weakref__属性。
class C3():
    __slots__ = 'who'
```

请将上述代码保存为weakref1.py，然后执行如下命令行：

```
$ python3 -i weakref1.py
```

接下来我们将基于该例子验证很多前面提到的关于弱引用的结论。

下面的语句说明了，可以为任何类型对象创建弱引用，并通过调用弱引用来获取该类型对象：

```
>>> wr = weakref.ref(typing.Any)
>>> wr
<weakref at 0x1086031a0; to '_AnyMeta' at 0x7fdbdec7ad90 (Any)>
>>> type(wr)
<class 'weakref'>
>>> weakref.ReferenceType
<class 'weakref'>
>>> wr()
typing.Any
>>>
```

下面的语句展示了如何为一个函数创建弱引用，并通过弱引用调用它：

```
>>> wr = weakref.ref(f1)
>>> wr
<weakref at 0x1085ab290; to 'function' at 0x1083745e0 (f1)>
>>> wr()
<function f1 at 0x1083745e0>
>>> wr()()
I am a function, my name is 'f1'.
>>>
```

下面的语句展示了如何为一个类创建弱引用，并通过弱引用调用它：

```
>>> wr = weakref.ref(C1)
>>> wr
<weakref at 0x10197f3d0; to 'type' at 0x7feb31e48c70 (C1)>
>>> wr()
<class '__main__.C1'>
>>> obj1 = wr()('obj1')
>>>
```

下面的语句展示了如何为一个方法创建弱引用，并通过弱引用调用它：

```
>>> wr = weakref.ref(obj1.m1)
>>> wr
<weakref at 0x10197f240; dead>
>>> wr()
>>> wr = weakref.ref(C1.m1)
>>> wr
<weakref at 0x101a20860; to 'function' at 0x101a0df80 (m1)>
>>> wr()
<function C1.m1 at 0x101a0df80>
>>> wr()(obj1)
I am 'obj1'.
>>>
```

注意其实我们只能通过实例访问到方法对象，而该方法对象是瞬时存在的，为其创建弱引用，调用弱引用时会得到None。而通过类访问一个函数属性时，得到的是一个函数，因此本质上是在为函数创建弱引用。

下面的语句说明，我们可以为一个类方法创建弱引用，但该弱引用其实无法使用：

```
>>> wr = weakref.ref(obj1.m2)
>>> wr
<weakref at 0x10197f240; dead>
>>> wr()
>>> wr = weakref.ref(C1.m2)
>>> wr
<weakref at 0x101a20860; dead>
>>> wr()
>>>
```

下面的语句展示了如何为一个具有__weakref__属性的类的实例创建弱引用，并通过弱引用访问它：

```
>>> wr = weakref.ref(obj1)
>>> wr
<weakref at 0x10197f240; to 'C1' at 0x101a28b50>
>>> wr()
<__main__.C1 object at 0x101a28b50>
>>> wr().m2()
My class is 'C1'.
>>>
```

下面的语句则说明了如何为一个静态方法创建弱引用，并通过弱引用调用它：

```
>>> obj2 = C2('obj2')
>>> wr = weakref.ref(obj2.m3)
>>> wr
<weakref at 0x101a20860; to 'function' at 0x101a26200 (m3)>
>>> wr()
<function C2.m3 at 0x101a26200>
>>> wr()()
I am 'C2'.
>>> wr = weakref.ref(C2.m3)
>>> wr
<weakref at 0x101a20860; to 'function' at 0x101a26200 (m3)>
>>> wr()
<function C2.m3 at 0x101a26200>
>>> wr()()
I am 'C2'.
>>>
```

注意对于静态方法来说，不论通过实例访问它，还是通过类访问它，得到的都是一个函数，因此本质上是在为函数创建弱引用。

下面的语句说明了，只要一个类具有__weakref__，就可以为它的实例创建弱引用；否则会抛出TypeError异常：

```
>>> obj3 = C3()
>>> wr = weakref.ref(obj2)
>>> wr
<weakref at 0x10197f240; to 'C2' at 0x101a28c90>
>>> wr()
<__main__.C2 object at 0x101a28c90>
>>> wr = weakref.ref(obj3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot create weak reference to 'C3' object
>>>
```

注意C2和C3都设置了__slots__属性，所以不能仅凭类定义语句中是否设置了__slots__来判断该类的实例是否能被弱引用。

从上面的结果可以看出，通过weakref.ref实例化出的ref对象虽然可以弱引用一个方法或类方法，但该ref对象必然是死亡的，没有意义。weakref提供了WeakMethod类来解决这个问题，其实例化语法为：

```
class weakref.WeakMethod(method)
```

其中method参数是一个方法或类方法。WeakMethod对象与ref对象的不同之处在于，它其实是实例或类的弱引用，但通过特殊的代码，使得被调用时会动态创建指定的方法对象。

下面的语句说明了如何通过WeakMethod类弱引用一个方法，可以看出WeakMethod对象本质上是对ref对象的包装：

```
>>> wm = weakref.WeakMethod(obj1.m1)
>>> type(wm)
<class 'weakref.WeakMethod'>
>>> wm
<weakref at 0x10175e5e0; to 'C1' at 0x101a28b50>
>>> wm()
<bound method C1.m1 of <__main__.C1 object at 0x101a28b50>>
>>> wm()()
I am 'obj1'.
>>>
```

注意方法必须通过实例访问，此时WeakMethod对象弱引用实例。

下面的语句说明了如何通过WeakMethod类弱引用一个类方法：

```
>>> wm = weakref.WeakMethod(obj1.m2)
>>> wm
<weakref at 0x1019cd000; to 'type' at 0x7feb31e48c70 (C1)>
>>> wm()
<bound method C1.m2 of <class '__main__.C1'>>
>>> wm()()
My class is 'C1'.
>>> wm = weakref.WeakMethod(C1.m2)
>>> wm
<weakref at 0x10175e5e0; to 'type' at 0x7feb31e48c70 (C1)>
>>> wm()
<bound method C1.m2 of <class '__main__.C1'>>
>>> wm()()
My class is 'C1'.
>>>
```

注意类方法既可以通过实例访问，也可以通过类访问，此时WeakMethod对象弱引用类。

上面的例子中，在创建弱引用时都没有指定回调函数。如果创建弱引用时指定了回调函数，则该回调函数被调用意味着同时满足下面两个条件：

- 被弱引用的对象即将被销毁。
- 该弱引用依然存在。

而该回调函数必须具有一个形式参数，它将被传入弱引用本身。弱引用通过__callback__属性记录了该回调函数。如果创建弱引用时没有指定回调函数，或者被弱引用的对象已经不存在，则__callback__属性将引用None。

需要强调的是，当上述回调函数被调用时，其被传入的弱引用已经死亡。因此回调函数能够利用被传入的弱引用做的唯一一件事，就是销毁该弱引用。如果在回调执行过程中抛出了异常，则该异常无法传递，与__del__中抛出异常的情形完全相同。

我们可以给一个对象创建任意多个弱引用，因此也能通过这些弱引用注册任意多个回调函数。当该对象被销毁时，会按照这些回调函数被注册的顺序的逆序依次调用它们。

由于一个对象可以具有多个弱引用，所以weakref提供了getweakrefcount()来获取一个对象的弱引用的数量，其语法为：

```
weakref.getweakrefcount(obj)
```

类似的，sys提供了表15-4列出的sys属性来获取一个对象的引用的数量，其语法为：

sys.getrefcount(*obj*)

需要强调，getweakrefcount()返回的整数就是该对象的弱引用数，而getrefcount()返回的整数是该对象的引用数加1，因为getrefcount()本身会为该对象提供一个引用。

表15-4. 对象的引用数相关sys属性

属性	说明
sys.getrefcount(*obj*)	返回对象obj的引用数。

一个对象的引用来自于其他对象的属性，访问它们会得到该对象本身。然而一个对象的弱引用来自于ref对象，能够取得这些ref对象是有意义的。因此weakref提供了getweakrefs()来取得一个对象的所有弱引用对应的ref对象，其语法为：

weakref.getweakrefs(*obj*)

它会返回由ref对象（以及后面会介绍的proxy对象）形成的列表。

最后，弱引用之间可以进行==和!=比较，这等价于先调用它们，然后比较获得的对象。注意已经死亡的弱引用总是相等的。

下面通过一个例子来验证上面介绍的技巧：

```
import weakref, sys

#定义第一个回调函数。
def cb1(wr):
    print("cb1 is called!")
    del wr
    raise RuntimeError()


#定义第二个回调函数。
def cb2(wr):
    print(f"cb2 is called!")
    del wr


#定义第三个回调函数。
def cb3(wr):
    print(f"cb3 is called!")
    del wr


#定义一个函数来显示一个对象的的引用数、弱引用数和ref对象。
def ref_stat(obj):
    print("Reference number: " + str(sys.getrefcount(obj)))
```

```
        print("WeakRef number: " + str(weakref.getweakrefcount(obj)))
        print("WeakRefs: " + repr(weakref.getweakrefs(obj)))
        print("")


    if __name__ == "__main__":
        #创建两个集合。
        st1 = {'a', 'b', 'c'}
        st2 = {'c', 'b', 'a'}
        print(f"st1 is {st1}")
        print(f"st2 is {st2}")
        print("")

        #为st1创建第一个弱引用:
        wr1 = weakref.ref(st1, cb1)
        print("wr1's callback: " + repr(wr1.__callback__))
        ref_stat(st1)

        #为st1创建第二个弱引用:
        wr2 = weakref.ref(st1, cb2)
        print("wr2's callback: " + repr(wr2.__callback__))
        ref_stat(st1)

        #为st1创建第三个弱引用:
        wr3 = weakref.ref(st1, cb3)
        print("wr3's callback: " + repr(wr3.__callback__))
        ref_stat(st1)

        #为st2创建一个弱引用:
        wr4 = weakref.ref(st2)
        print("wr4's callback: " + repr(wr4.__callback__))
        print("")


        #验证弱引用之间的比较:
        print("wr1 == wr2 == wr3: " + str(wr1 == wr2 == wr3))
        print("wr1 == wr4: " + str(wr1 == wr4))
        print("delete st2!")
        del st2
        print("wr1 == wr4: " + str(wr1 == wr4))
        print("")

        #删除st1的第二个弱引用，然后删除st1本身。
        print("delete wr2!")
        del wr2
        ref_stat(st1)
        print("delete st1!")
        del st1
```

请将上述代码保存为weakref2.py，然后通过如下命令行验证：

```
    $ python3 weakref2.py
    st1 is {'a', 'b', 'c'}
    st2 is {'a', 'b', 'c'}

    wr1's callback: <function cb1 at 0x1016607c0>
    Reference number: 3
    WeakRef number: 1
    WeakRefs: [<weakref at 0x1016ec8b0; to 'set' at 0x1016e2260>]

    wr2's callback: <function cb2 at 0x1016e8860>
```

```
      Reference number: 3
      WeakRef number: 2
      WeakRefs: [<weakref at 0x1016ec860; to 'set' at 0x1016e2260>, <weakref at
0x1016ec8b0; to 'set' at 0x1016e2260>]

      wr3's callback: <function cb3 at 0x1016e87c0>
      Reference number: 3
      WeakRef number: 3
      WeakRefs: [<weakref at 0x101714b30; to 'set' at 0x1016e2260>, <weakref at
0x1016ec860; to 'set' at 0x1016e2260>, <weakref at 0x1016ec8b0; to 'set' at
0x1016e2260>]

      wr4's callback: None

      wr1 == wr2 == wr3: True
      wr1 == wr4: True
      delete st2!
      wr1 == wr4: False

      delete wr2!
      Reference number: 3
      WeakRef number: 2
      WeakRefs: [<weakref at 0x101714b30; to 'set' at 0x1016e2260>, <weakref at
0x1016ec8b0; to 'set' at 0x1016e2260>]

      delete st1!
      cb3 is called!
      cb1 is called!
      Exception ignored in: <function cb1 at 0x1016607c0>
      Traceback (most recent call last):
        File "/Users/wwy/weakref2.py", line 7, in cb1
          raise RuntimeError()
          ^^^^^^^^^^^^^^^^^^^
      RuntimeError:
```

针对该结果需要额外提醒的是，在通过ref_stat()显示st1的引用数时，之所以会得到3，是因为标识符“st1”，形式参数“obj”和sys.getrefcount()分别提供了一个引用。

一个对象的弱引用是否是可哈希的，取决于该对象是否是可哈希的。但要注意的是，如果我们先对一个可哈希的弱引用调用hash()，然后再删除了被它弱引用的对象，该弱引用依然是可哈希的，且哈希值等于该对象的哈希值。但如果我们先删除一个可哈希的弱引用所弱引用的对象，再对该弱引用调用hash()，则会抛出TypeError，而不是得到None的哈希值。下面的例子验证了上述论断：

```
      $ python3

      >>> import weakref
      >>> class D(dict):
      ...     pass
      ...
      >>> o1 = D()
      >>> class C:
      ...     pass
      ...
      >>> o2 = C()
      >>> o3 = C()
```

```
>>> hash(o1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'D'
>>> hash(o2)
270239201
>>> hash(o3)
270239209
>>>
>>> wr = weakref.ref(o1)
>>> hash(wr)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'D'
>>>
>>> wr = weakref.ref(o2)
>>> hash(wr)
270239201
>>> del o2
>>> wr()
>>> hash(wr)
270239201
>>>
>>> wr = weakref.ref(o3)
>>> del o3
>>> wr()
>>> hash(wr)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: weak object has gone away
>>>
```

注意该例子用到了这样一个技巧：内置类型list和dict的实例不支持弱引用，但它们派生的子类的实例支持弱引用，而这些实例都是不可哈希的。

直接使用ref对象或WeakMethod对象比较麻烦。每次使用它们都需要按照如下范式判断被弱引用的对象是否存在：

```
#wr_or_wm代表一个ref对象或WeakMethod对象。
o = wr_or_wm()
if o is None:
    ... codes for weakly referenced object doesn't exist ...
else:
    ... codes for weakly referenced object exists ...
```

注意不能将上述范式改写为：

```
if wr_or_wm() is None:
    ... codes for weakly referenced object doesn't exist ...
else:
    ... codes for weakly referenced object exists ...
```

因为这在多线程环境下会导致产生竞争条件：在调用完wr_or_wm()后立即发生了线程切换，而wr_or_wm()返回的对象没有被标识符o引用，有可能引用数为0。

上述范式不太符合正常的编程习惯，因此weakref提供了proxy()函数来创建使用弱引用的代理，其语法为：

weakref.proxy(*obj*[, *callback*])

其obj参数和callback参数的含义与在weakref.ref()中相同，但该函数返回的是一个代理对象：当obj可调用时类型为weakref.CallableProxyType，即可调用代理；当obj不可调用时类型为weakref.ProxyType，即不可调用代理。可以用weakref.ProxyTypes判断一个标识符是否是一个代理，它等价于(weakref.ProxyType, weakref.CallableProxyType)元组。大体上，可以将代理视为该对象的另一个标识符，只不过该标识符使用的是弱引用，当该对象不再存在时对该标识符的访问将导致抛出ReferenceError异常。此外，不论该对象是否可哈希，其代理都是不可哈希的。下面的例子说明了如何使用代理：

```
$ python3

>>> import weakref
>>> fs = frozenset({'a', 'b', 'c'})
>>> def gen():
...     for i in range(3):
...         yield i
...
>>>
>>> weakref.ProxyType
<class 'weakproxy'>
>>> weakref.CallableProxyType
<class 'weakcallableproxy'>
>>> weakref.ProxyTypes
(<class 'weakproxy'>, <class 'weakcallableproxy'>)
>>>
>>> px1 = weakref.proxy(fs)
>>> type(px1)
<class 'weakproxy'>
>>> type(px1) in weakref.ProxyTypes
True
>>> px1
<weakproxy at 0x10e246ac0 to frozenset at 0x10e40eea0>
>>> px1.isdisjoint({'d', 'e'})
True
>>> px1.issuperset({'a'})
True
>>> del fs
>>> px1
<weakproxy at 0x10e246ac0 to NoneType at 0x10de59680>
>>> px1.issubset({'a'})
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ReferenceError: weakly-referenced object no longer exists
>>>
>>> px2 = weakref.proxy(gen)
>>> type(px2)
<class 'weakcallableproxy'>
>>> type(px2) in weakref.ProxyTypes
```

```
True
>>> px2
<weakproxy at 0x10e289670 to function at 0x10e45b240>
>>> g = px2()
>>> next(g)
0
>>> next(g)
1
>>> next(g)
2
>>> next(g)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>> del gen
>>> px2
<weakproxy at 0x10e289670 to NoneType at 0x10de59680>
>>> px2()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ReferenceError: weakly-referenced object no longer exists
>>>
```

需要强调的是，一个proxy对象在逻辑上等同于一个弱引用，会使getweakrefcount()的返回值增加1，并会出现在getweakrefs()返回的列表中。

以上就是weakref模块提供的低层级API。它们说明了弱引用的原理，并支持一些高级应用。举例来说，当我们创建复杂的数据结构（例如一个图）时，为了避免循环引用，会大量使用弱引用。然而对大多数Python编程者来说，可以不直接与弱引用打交道，因为weakref模块还提供了高层级API，即三种使用弱引用的容器类型——WeakSet、WeakKeyDictionary和WeakValueDictionary，以及用于创建“终结器（finalizers）”的finalize类。下面依次讨论这些高层级API。

WeakSet是一种特殊的集合，其实例化语法为：

```
class weakref.WeakSet([iterable])
```

WeakSet对象支持集合的一切操作，两者的区别在于，前者通过弱引用访问其内的元素，因此不会增加其元素的引用数。当WeakSet对象的某元素引用数变为0时，将自动从WeakSet对象中删除。但要注意，当迭代通过iterable参数指定的可迭代对象时，获得的对象必须是支持弱引用的。如果省略了iterable，则会得到一个空的WeakSet对象。下面的例子说明了如何使用WeakSet：

```
$ python3

>>> import weakref
>>> class C:
...     pass
...
```

```
      >>> o1 = C()
      >>> o2 = C()
      >>> o3 = C()
      >>> ws = weakref.WeakSet([o1, o2, o3])
      >>> ws
      {<weakref at 0x10feaf1a0; to 'C' at 0x10fe973d0>, <weakref at 0x10feaf100;
to 'C' at 0x10fe97450>, <weakref at 0x10fcce3e0; to 'C' at 0x10fe97350>}
      >>> o1 in ws
      True
      >>> 0 in ws
      False
      >>> ws.intersection({o1, o3})
      {<weakref at 0x10feaf290; to 'C' at 0x10fe97350>, <weakref at 0x10feaf240;
to 'C' at 0x10fe97450>}
      >>> ws.difference({o1, o3})
      {<weakref at 0x10fc88ef0; to 'C' at 0x10fe973d0>}
      >>> {o1, o3}.difference(ws)
      set()
      >>> del o1
      >>> ws
      {<weakref at 0x10feaf1a0; to 'C' at 0x10fe973d0>, <weakref at 0x10feaf100;
to 'C' at 0x10fe97450>}
      >>> del o2
      >>> ws
      {<weakref at 0x10feaf100; to 'C' at 0x10fe97450>}
      >>> del o3
      >>> ws
      set()
      >>>
```

WeakKeyDictionary是一种特殊的字典，其实例化语法为：

```
class weakref.WeakKeyDictionary([iterable])
```

其iterable参数的含义与在dict()中相同，但不支持任何基于关键字对应的实际参数。如果省略了iterable，则会得到一个空的WeakKeyDictionary对象。WeakKeyDictionary对象支持字典的一切操作（但items()、keys()和values()返回的是生成器而非视图），两者的区别在于，前者通过弱引用记录其内的键，因此不会增加键的引用数。当一个键的引用数变为0时，相应键值对将自动从WeakKeyDictionary对象中删除。WeakKeyDictionary对象中的键不仅要是可哈希的，还必须支持弱引用。下面是一个使用WeakKeyDictionary的例子：

```
      $ python3

      >>> import weakref
      >>> class C:
      ...     pass
      ...
      >>> o1 = C()
      >>> o2 = C()
      >>> o3 = C()
      >>> wkd = weakref.WeakKeyDictionary([(o1, 1), (o2, 2), (o3, 3)])
      >>> wkd[o1]
      1
      >>> wkd.items()
      <generator object WeakKeyDictionary.items at 0x103a18a40>
```

```
>>> wkd.keys()
<generator object WeakKeyDictionary.keys at 0x1038a6d40>
>>> wkd.values()
<generator object WeakKeyDictionary.values at 0x1038a75b0>
>>> for k in wkd:
...     print(f"{k}: {wkd[k]}")
...
<__main__.C object at 0x103a25d90>: 1
<__main__.C object at 0x103a25e10>: 2
<__main__.C object at 0x103a25e90>: 3
>>> del o1
>>> for k in wkd:
...     print(f"{k}: {wkd[k]}")
...
<__main__.C object at 0x103a25e10>: 2
<__main__.C object at 0x103a25e90>: 3
>>>
```

WeakKeyDictionary还提供了一个通往weakref模块低层级API的接口，即keyrefs()属性。调用keyrefs()会返回一个可迭代对象，对其进行迭代将得到WeakKeyDictionary对象在keyrefs()被调用时保存的所有ref对象。然而需要强调，当我们使用这些ref对象时，它可能已经死亡，而非弱引用某个键，所以必须严格按照前面提到的范式来使用它们。下面是一个使用keyrefs()的例子：

```
import weakref

#定义两个将被弱引用的可哈希对象。
fs1 = frozenset({'a'})
fs2 = frozenset({'b'})


#该函数显示WeakKeyDictionary对象的键值对。
def show_kv_pairs(wkd):
    #调用keyrefs()以获得所有ref对象。
    for wr in wkd.keyrefs():
        o = wr()
        if o is None:
            pass
        else:
            print(f"{o}: {wkd[o]}")


#创建一个空的WeakKeyDictionary对象。
wkd1 = weakref.WeakKeyDictionary()

#将键值对插入wkd1。
wkd1[fs1] = 1
wkd1[fs2] = 2

#创建wkd1的拷贝。
wkd2 = weakref.WeakKeyDictionary(wkd1)

#显示wkd1和wkd2的初始状态。
print("initially")
print("wkd1:")
show_kv_pairs(wkd1)
print("wkd2:")
show_kv_pairs(wkd2)
print("")
```

```
#删除fs1，然后显示wkd1和wkd2现在的状态。
print("delete {fs1}")
del fs1
print("wkd1:")
show_kv_pairs(wkd1)
print("wkd2:")
show_kv_pairs(wkd2)
print("")

#删除fs2，然后显示wkd1和wkd2现在的状态。
print("delete {fs2}")
del fs2
print("wkd1:")
show_kv_pairs(wkd1)
print("wkd2:")
show_kv_pairs(wkd2)
print("")
```

请将上述代码保存为weakref3.py，然后通过如下命令行验证：

```
$ python3 weakref3.py
initially
wkd1:
frozenset({'a'}): 1
frozenset({'b'}): 2
wkd2:
frozenset({'a'}): 1
frozenset({'b'}): 2

delete {fs1}
wkd1:
frozenset({'b'}): 2
wkd2:
frozenset({'b'}): 2

delete {fs2}
wkd1:
wkd2:
```

WeakValueDictionary也是一种特殊的字典，其实例化语法为：

```
class weakref.WeakValueDictionary([iterable])
```

它与WeakKeyDictionary的区别仅在于，通过弱引用记录的是值而不是键，因此只需要值支持弱引用。当一个值的引用数变为0时，相应键值对同样将自动从WeakKeyDictionary对象中删除。下面是一个使用WeakValueDictionary的例子：

```
$ python3

>>> import weakref
>>> class C:
...     pass
...
```

```
>>> o1 = C()
>>> o2 = C()
>>> o3 = C()
>>> wvd = weakref.WeakValueDictionary([(1, o1), (2, o2), (3, o3)])
>>> wvd[1]
<__main__.C object at 0x10fadddd0>
>>> wvd.items()
<generator object WeakValueDictionary.items at 0x10fad0a40>
>>> wvd.keys()
<generator object WeakValueDictionary.keys at 0x10f95ed40>
>>> wvd.values()
<generator object WeakValueDictionary.values at 0x10f95f5b0>
>>> for k in wvd:
...     print(f"{k}: {wvd[k]}")
...
1: <__main__.C object at 0x10fadddd0>
2: <__main__.C object at 0x10fadde50>
3: <__main__.C object at 0x10fadded0>
>>> del o1
>>> for k in wvd:
...     print(f"{k}: {wvd[k]}")
...
2: <__main__.C object at 0x10fadde50>
3: <__main__.C object at 0x10fadded0>
>>>
```

WeakValueDictionary同样提供了一个通往weakref模块低层级API的接口，即valuerefs()属性。valuerefs()的用法与keyrefs()类似。下面是一个使用valuerefs()的例子：

```
import weakref

#定义两个将被弱引用的不可哈希对象。
st1 = {'a'}
st2 = {'b'}


#该函数显示WeakValueDictionary对象的值。
def show_values(wvd):
    #调用valuerefs()以获得所有ref对象。
    for wr in wvd.valuerefs():
        o = wr()
        if o is None:
            pass
        else:
            print(o)


#创建一个空的WeakValueDictionary对象。
wvd1 = weakref.WeakValueDictionary()

#将键值对插入wvd1。
wvd1[1] = st1
wvd1[2] = st2

#创建wvd1的拷贝。
wvd2 = weakref.WeakValueDictionary(wvd1)

#显示wvd1和wvd2的初始状态。
print("initially")
print("wvd1:")
show_values(wvd1)
```

```
print("wvd2:")
show_values(wvd2)
print("")

#删除st1，然后显示wvd1和wvd2现在的状态。
print("delete {st1}")
del st1
print("wvd1:")
show_values(wvd1)
print("wvd2:")
show_values(wvd2)
print("")

#删除st2，然后显示wvd1和wvd2现在的状态。
print("delete {st2}")
del st2
print("wvd1:")
show_values(wvd1)
print("wvd2:")
show_values(wvd2)
print("")
```

请将上述代码保存为weakref4.py，然后通过如下命令行验证：

```
$ python3 weakref4.py
initially
wvd1:
{'a'}
{'b'}
wvd2:
{'a'}
{'b'}

delete {st1}
wvd1:
{'b'}
wvd2:
{'b'}

delete {st2}
wvd1:
wvd2:
```

最后，让我们讨论终结器。在第5章介绍过，通过给自定义类添加__del__魔术属性，可以使得该类的实例在被销毁前自动执行一些操作。然而使用这一技术的前提是能够修改类定义本身。有些时候我们需要使用第三方编写的模块，并不期望修改该模块的代码，但又希望该模块定义的类的实例在销毁前自动执行一些操作。在这种情况下，一个解决方案是为这些实例创建ref对象或proxy对象，并在创建它们时指定回调函数。然而该方案要用到weakref模块的底层API，另外必须确保ref对象或proxy对象自身的引用数在该对象被销毁前不为0，而在该对象被销毁后必须手工删除它们以节省内存。终结器是一个更好的解决方案，省去了我们手工维护ref对象或proxy对象的麻烦。

终结器是finalize类（类名同样不符合PEP 8推荐）的实例，其实例化语法为：

class **weakref.finalize(*obj*, *func*, /, **args*, ***kwargs*)**

其中obj是该终结器被注册到的对象，func是回调函数的函数名，args和kwargs则是传给回调函数的参数。终结器被创建后，在obj被销毁时会自动调用func(*args, **kwargs)。需要强调的是，obj本身会给注册的终结器提供一个引用，而在obj被销毁后该引用也自然消失，使得我们不需要将新创建的终结器赋值给额外的标识符。

必须要强调，func、args和kwargs不能以任何直接或间接的方式提供对obj的引用，否则obj的引用数将永远不会为0。另外，func也不能是obj的属性，否则当它被调用时就已经不存在了。如果在func被调用的过程中抛出了某个异常，则该异常也不能被传递，就好像执行__del__的过程中抛出异常一样。

下面的例子给一个集合注册了一个匿名终结器，然后删除该集合：

```
$ python3

>>> import weakref, time
>>> def cb():
...     print("finalize an object ...")
...     time.sleep(1)
...     print("finished!")
...
>>> st = {'a', 'b', 'c'}
>>> weakref.finalize(st, cb)
<finalize object at 0x104e86620; for 'set' at 0x104fb75a0>
>>> del st
finalize an object ...
finished!
>>>
```

当然，如果我们将新创建的终结器赋值给某个标识符，则可以通过该标识符对终结器进行更多的操作。终结器在内部是通过ref对象实现的，所以也有存活和死亡两种状态，可以通过alive属性查看，该属性引用True时表示终结器存活，引用False时表示终结器死亡。

终结器是可调用的。当终结器处于存活状态时，调用它导致调用func(*args, **kwargs)，并以其返回值作为终结器的返回值，但同时终结器会变成死亡状态。而当终结器处于死亡状态时，调用它直接得到None。终结器注册到的对象被销毁时，如果终结器已经死亡，则不会被自动调用。换句话说，一个终结器的回调函数func至多被调用一次。

下面的例子给自定义类C的实例o注册了一个终结器fin，然后显式调用它：

```
$ python3

>>> import weakref
>>> class C:
...     pass
...
```

```
>>> o = C()
>>> def f(msg):
...     print(msg)
...
>>> fin = weakref.finalize(o, f, "o is deleted!")
>>> fin.alive
True
>>> fin()
o is deleted!
>>> fin.alive
False
>>> fin()
>>> del o
>>>
```

如果不显式调用一个终结器，那么该终结器就会在其注册到的对象被销毁时自动被调用。如果一个终结器注册到的对象直到脚本运行结束时才被销毁，则该终结器是否会被调用取决于它的atexit属性：当该属性引用True时，该终结器会被执行；当该属性引用False时，该终结器不会被执行。atexit属性默认引用True。当然，如果运行Python解释器的进程意外崩溃，那么尚未被调用的终结器不论atexit属性引用哪个布尔值都不会被调用。

下面的例子给自定义类C注册了两个终结器fin1和fin2，将fin2的atexit属性设置为False，然后调用exit()结束了脚本的执行：

```
$ python3

>>> import weakref
>>> class C:
...     pass
...
>>> def f(*, name):
...     print(f"{name} is called.")
...
>>> fin1 = weakref.finalize(C, f, name="fin1")
>>> fin2 = weakref.finalize(C, f, name="fin2")
>>> fin1.atexit
True
>>> fin2.atexit
True
>>> fin2.atexit = False
>>> exit()
fin1 is called.
```

如果想取消一个终结器的注册，则可以调用该终结器的detach()属性。如果该终结器处于存活状态，则会返回元组(obj, func, args, kwargs)以说明该终结器的基本信息，并将终结器的状态设置为死亡。如果该终结器处于死亡状态，则会返回None。而终结器的peek()属性与detach()属性的唯一区别是不会将终结器的状态设置为死亡。下面的例子说明了这两个属性的作用：

```
$ python3

>>> import weakref
>>> fs = frozenset({'abc'})
```

```
      >>> def f(*args, **kwargs):
      ...     pass
      ...
      >>> fin = weakref.finalize(fs, f, 0, 1, 2, a='x', b='y')
      >>> fin.alive
      True
      >>> fin.peek()
      (frozenset({'abc'}), <function f at 0x10574b240>, (0, 1, 2), {'a': 'x',
'b': 'y'})
      >>> fin.alive
      True
      >>> fin.detach()
      (frozenset({'abc'}), <function f at 0x10574b240>, (0, 1, 2), {'a': 'x',
'b': 'y'})
      >>> fin.alive
      False
      >>> fin.detach()
      >>>
```

至此，本书已经完成了对Python语法的讨论。后面两章将讨论Python编程者必须掌握的一些开发技巧。