第5章. 类

虽然在Python中一切皆对象，但如果在编写Python脚本时只定义和调用函数，那么依然是在以过程化的方式编程。对于绝大多数中小规模的项目来说，过程化的方式就是最好的解决方案，因为它能够让代码尽可能精简，且更易懂。然而随着项目的规模越来越大，参与的人越来越多，因调用函数时传入了不合适的实际参数导致难以发现的错误的概率会越来越高。这一问题主要是因为项目参与者之间沟通不畅，错误理解对方编写函数的使用方式（或执行细节）导致的。

解决该问题的一种办法是编写函数时提供清晰且规范的文档（在第6章会进一步讨论），然而该办法存在薄弱性：文档写得再好，如果函数的使用者不去阅读，也就不能发挥任何作用。解决该问题的另一种办法是以面向对象的方式编程，使发生此类错误的概率尽可能小。面向对象技术就是本章探讨的主题。

5-1. 定义类的基本方法

（教程：9.3.1）
（语言参考手册：3.2、8.8）

面向对象技术以“类”为核心，其本质是将类型与属于该类型的对象可执行的操作绑定起来，以防止因对某对象执行不合适的操作而导致的错误。第3章已经说明，在Python中类是通过类对象表示的，它们与类型对象在逻辑上等同。

在Python中定义类的基本方法是使用类定义语句，该语句的语法为：

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

与函数定义语句类似，类定义语句被执行后会在内存空间中创建一个类对象，该对象默认会一直存在到脚本运行结束。clsname为引用该类对象的标识符，亦即该类的“类名（class name）”，相应字符串会被该类对象的__name__特殊属性引用。argument_list代表的是“类参数列表（class argument lists）”，注意它在逻辑上等同于函数调用中的实际参数列表，而非函数定义中的形式参数列表，其作用是设置类的继承关系（将在本章后面讨论）和该类所属于的元类（将在第15章讨论）。与函数定义必须包含圆括号不同，类定义中整个“()”和其内的类参数列表都是可选的。由于object是所有类的基类，所以：

```
class Foo:
    pass
```

等价于：

```
class Foo(object):
    pass
```

类定义语句中的suite代表类的“类体（class body）”。与其他编程语言（例如C++和Java）不同，Python中的类体能够包含各种语句（包括分支和循环）。类体与函数体的唯一区别是类体中不能有return语句，因为类体被执行的结果总是创建一个类对象，没有返回值。此外，函数定义语句被执行时其函数体会被转换为一个代码对象，但该代码对象并不会立刻被执行，而是在函数调用时才被执行。而类定义语句被执行时其类体也会被转换为一个代码对象，但该代码对象后立刻被执行然后丢弃。换句话说，在脚本运行过程中函数体通常会被执行很多次，但类体只会被执行一次。

具体来说，类定义语句在被执行的过程中并不涉及__call__魔术属性，但依然会给函数栈添加一个帧对象，该帧对象的f_code属性引用类体转换成的代码对象，但f_locals引用的变量字典不会储存类参数列表中的标识符（它们是嵌入代码对象的）。类体被执行过程中，所有新定义的标识符都会储存到f_locals引用的变量字典中，而当类体执行结束后才会创建类对象。该类对象的__dict__会引用一个新创建的字典或该字典的只读代理，而该字典会被用f_locals属性引用的变量字典中的键值对填充，进而使类体中定义的标识符被保留下来。类参数列表只被用于设置新建类对象的特殊属性__class_、__bases__和__mro__。

综上所述，编写类体的关键在于控制执行该类体后新定义了哪些标识符，它们将被储存到相应类对象的变量字典中。这些标识符会成为该类对象的实例属性，进而成为该类实例的类属性。一般而言，类体是由若干赋值语句和若干函数定义语句构成的，前者为类对象添加非函数属性，后者为类对象添加函数属性。下面重温第3章Point2D.py中的例子：

```
class Point2D:
    #类属性dimension。
    dimension = 2

    #在初始化时设置了实例属性x和y。
    def __init__(self, x, y):
        self.x = x
        self.y = y

    #类属性move。
    def move(self, dx, dy):
        self.x += dx
        self.y += dy

    #类属性location。
    def location(self):
        print("(" + str(self.x) + ", " + str(self.y) + ")")
```

在第3章我们对该类定义语句没有仔细分析，现在可以知道它定义了一个名为“Point2D”的类，该类的直接基类只有object，该类的实例共享非函数属性dimension以及函数属性__init__、move()和location()。

请用如下命令行和语句查看类对象Point2D的特殊属性__dict__、__class__、__bases__和__mro__：

```
$ python3 -i Point2D.py

>>> Point2D.__dict__
mappingproxy({'__module__': '__main__', 'dimension': 2, '__init__':
<function Point2D.__init__ at 0x105d0c790>, 'move': <function Point2D.move at
0x105d0c820>, 'location': <function Point2D.location at 0x105d0c8b0>, '__dict__':
<attribute '__dict__' of 'Point2D' objects>, '__weakref__': <attribute
'__weakref__' of 'Point2D' objects>, '__doc__': None})
>>> Point2D.__class__
<class 'type'>
>>> Point2D.__bases__
(<class 'object'>,)
>>> Point2D.__mro__
(<class '__main__.Point2D'>, <class 'object'>)
>>>
```

注意除了类体中定义的标识符“dimension”、“__init__”、“move”和“location”之外，类对象的变量字典还包括如下键：“__doc__”、“__module__”、“__dict__”和“__weakref__”，但前三者等同于该类对象的同名特殊属性，__weakref__是一个用于支持弱引用的数据描述器（将在第15章详细讨论），都不会成为该类实例的类属性。

Point2D例子中，类体仅包含赋值语句和函数定义语句。下面给出一个类体中包含分支的类定义例子：

```
from random import randrange


class BranchAttr:
    #生成一个0～9之间的随机整数。
    n = randrange(0, 10)

    #当随机整数是0、1、2、3、4时定义star()属性。
    if n<5:
        def star():
            return '*'
    #当随机整数是5、6、7、8、9时定义dollar()属性。
    else:
        def dollar():
            return '$'

    #删除引用随机整数的标识符，避免它成为类的属性。
    del n
```

该例子从标准库的random模块导入了randrange()函数，以生成一个0～9内的随机整数，但本书不讨论random模块。请将上述代码保存为BranchAttr.py，然后多次通过如下命令行和语句验证BranchAttr类将随机地具有star()属性或dollar()属性：

```
$ python3 -i BranchAttr.py

>>> BranchAttr.__dict__
mappingproxy({'__module__': '__main__', 'star': <function RandomAttr.star
at 0x1057e1ea0>, '__dict__': <attribute '__dict__' of 'RandomAttr' objects>,
'__weakref__': <attribute '__weakref__' of 'RandomAttr' objects>, '__doc__':
None})      #这里的输出中也可能没有'star'，而有'dollar'。
>>>
```

由于类体的执行过程与函数体的执行过程没有本质不同，所以在类体中也能访问全局变量（乃至它被嵌套入的函数体的本地变量）。下面给出一个这样的类定义例子，注意该例子的类体中还包含了循环：

```
#通过标准输入获得一个正整数。
n = input("请输入一个正整数: ")
n = int(n)
if n < 1:
    n = 1


class LoopAttr:
    #将n声明为全局变量，以避免它成为类的属性。
    global n

    #创建属性a1、a2、...、an。
    #分别将它们赋值1、2、...、n。
    while n > 0:
        exec('a'+str(n)+' = '+str(n))
        n = n-1
```

该例子在类体内通过global关键字将标识符n声明为全局变量，并通过内置函数exec()将一个拼接好的字符串当成赋值语句来执行（这些会在第6章详细讨论）。请将上述代码保存为LoopAttr.py，然后多次通过如下命令行和语句验证LoopAttr类将具有属性a1、a2、……、an：

```
$ python3 -i LoopAttr.py
请输入一个正整数: 5  #这里也可以输入其他正整数。

>>> LoopAttr.__dict__
mappingproxy({'__module__': '__main__', 'a5': 5, 'a4': 4, 'a3': 3, 'a2': 2,
'a1': 1, '__dict__': <attribute '__dict__' of 'LoopAttr' objects>, '__weakref__':
<attribute '__weakref__' of 'LoopAttr' objects>, '__doc__': None})
>>>
```

最后，PEP 8建议在类定义语句前后都添加两个空行。

5-2. 定义类的特殊方法

（标准库：内置函数、types）

使用类定义语句是定义类的基本方法，但还有两种定义类的特殊方法。第一种方法是使用内置函数type()，此时type()的语法与第3章中介绍的不同：

```
class type(name, bases, dict, **kwds)
```

type()的这种用法返回一个新创建的类对象，各参数都用于对该类对象进行设置：

1. 通过name参数传入的字符串将被类对象的__name__特殊属性引用，成为类名。
2. 通过bases参数传入的以其他类对象为成员的元组将被该类对象的__bases__特殊属性引用，其成员成为该类的直接基类。该类对象的__mro__特殊属性则会根据__bases__自动设置。如果bases参数被传入空元组，则该类的直接基类只有object。
3. 通过dict参数传入的映射将用于设置该类对象的变量字典，进而决定该类对象的实例属性。如果dict参数被传入空映射，则该类对象没有实例属性。
4. 通过kwds传入的所有基于关键字对应的实际参数都用于进行除元类之外的设置。

最后，该类对象的__class__特殊属性总是引用type()，亦即以type()为元类。

下面用type()来定义Point2D类：

```
#定义将作为Point2D类的__init__属性的函数。
def f1(self, x, y):
    self.x = x
    self.y = y


#定义将作为Point2D类的move属性的函数。
def f2(self, dx, dy):
    self.x += dx
    self.y += dy


#定义将作为Point2D类的location属性的函数。
def f3(self):
    print("(" + str(self.x) + ", " + str(self.y) + ")")


#定义Point2D类。
Point2D = type('Point2D', (), {
    'dimension': 2,
    '__init__': f1,
    'move': f2,
    'location': f3})


#修改函数属性的名字和限定名字。
Point2D.__init__.__name__ = '__init__'
Point2D.__init__.__qualname__ = 'Point2D.__init__'
Point2D.move.__name__ = 'move'
Point2D.move.__qualname__ = 'Point2D.move'
Point2D.location.__name__ = 'location'
Point2D.location.__qualname__ = 'Point2D.location'
```

```
    #删除不再需要的标识符。
    del f1
    del f2
    del f3
```

请将上述代码保存为Point2D_type.py，然后通过如下命令行和语句来验证它的功能与Point2D.py相同：

```
    $ python3 -i Point2D_type.py

    >>> Point2D.__dict__
    mappingproxy({'dimension': 2, '__init__': <function Point2D.__init__ at
0x1021efeb0>, 'move': <function Point2D.move at 0x102398790>, 'location':
<function Point2D.location at 0x102398820>, '__module__': '__main__', '__dict__':
<attribute '__dict__' of 'Point2D' objects>, '__weakref__': <attribute
'__weakref__' of 'Point2D' objects>, '__doc__': None})
    >>> Point2D.__class__
    <class 'type'>
    >>> Point2D.__bases__
    (<class 'object'>,)
    >>> Point2D.__mro__
    (<class '__main__.Point2D'>, <class 'object'>)
    >>>
```

下面将通过type()定义类与用类定义语句定义类做比较。首先，name参数仅能够设置__name__属性，type()返回的类对象必须赋值给另一个标识符才能使用，为了与类定义语句中的clsname一致，该标识符必须与name参数传入的字符串相同。然后，bases参数与kwds参数合在一起才与类定义语句中的argument_list对应，然而用type()创建的类的元类只能是type()，所以通过kwds参数传入的metaclass设置会被忽略。最后，通过dict参数传入的映射取代了类定义语句中的类体，映射中的每个键值对都相当于一条赋值语句或一条函数定义语句，而其他语句无法通过映射表示。

此外需要强调的是，给通过type()定义的类添加函数属性的方法有两种：

1. 让传入dict参数的映射的相应健值对的值取lambda表达式。
2. 先定义一个函数，然后让让传入dict参数的映射的相应健值对的值取该函数的函数名。

不论用哪种方法，在类被创建后都需手工修改函数属性的__name__和__qualname__，以使它们引用正确的名字。

第二种方法是使用标准库中的types模块定义的new_class()函数，其语法为：

```
    class types.new_class(name, bases=(), kwds=None,
exec_body=None)
```

new_class()同样返回一个新创建的类对象，并基于传入的实际参数对该类对象进行设置，但与type()的第二种用法存在如下不同：

1. bases参数以空元组作为默认实参值，表示类对象默认以object类为直接基类。
2. kwds参数接收的映射用于对类进行设置。该映射可以通过“metaclass”键指定元类，因此new_class()创建的类不局限于以type()作为元类。

3. exec_body参数接收一个回调函数以设置类对象的变量字典，该回调函数的函数体中可以包含任何语句，因此比type()要灵活得多。

使用new_class()的难点在于给exec_body参数传入合适的回调函数：该回调函数必须仅具有一个参数ns（ns是约定俗成的标识符，理论上可以替换成其他标识符），当函数体被执行时ns引用的是一个空字典，而函数体的功能就是填充它，其返回值应设置为None（返回其他的对象也不会有什么影响）。而回调函数返回后，被其填充的字典将成为新建类对象的变量字典。

下面用new_type()来定义Point2D类：

```
import types


#定义将作为Point2D类的__init__属性的函数。
def f1(self, x, y):
    self.x = x
    self.y = y


#定义将作为Point2D类的move属性的函数。
def f2(self, dx, dy):
    self.x += dx
    self.y += dy


#定义将作为Point2D类的location属性的函数。
def f3(self):
    print("(" + str(self.x) + ", " + str(self.y) + ")")


#定义传入new_class()的exec_body参数的回调函数。
def eb(ns):
    ns['dimension'] = 2
    ns['__init__'] = f1
    ns['move'] = f2
    ns['location'] = f3


#定义Point2D类。
Point2D = types.new_class('Point2D', exec_body=eb)


#修改类所属的模块。
Point2D.__module__ = '__main__'

#修改函数属性的名字和限定名字。
Point2D.__init__.__name__ = '__init__'
Point2D.__init__.__qualname__ = 'Point2D.__init__'
Point2D.move.__name__ = 'move'
Point2D.move.__qualname__ = 'Point2D.move'
Point2D.location.__name__ = 'location'
Point2D.location.__qualname__ = 'Point2D.location'

#删除不再需要的标识符。
del f1
del f2
del f3
del eb
```

请将上述代码保存为Point2D_new_class.py，然后通过如下命令行和语句来验证它的功能与Point2D.py相同：

```
$ python3 -i Point2D_new_class.py

>>> Point2D.__dict__
mappingproxy({'dimension': 2, '__init__': <function Point2D.__init__ at
0x1021efeb0>, 'move': <function Point2D.move at 0x102398790>, 'location':
<function Point2D.location at 0x102398820>, '__module__': '__main__', '__dict__':
<attribute '__dict__' of 'Point2D' objects>, '__weakref__': <attribute
'__weakref__' of 'Point2D' objects>, '__doc__': None})
>>> Point2D.__class__
<class 'type'>
>>> Point2D.__bases__
(<class 'object'>,)
>>> Point2D.__mro__
(<class '__main__.Point2D'>, <class 'object'>)
>>>
```

从上面的例子可知，通过types.new_class()定义类的过程与通过type()定义类的过程基本相同。但除了需额外定义一个回调函数外，还要注意通过types.new_class()创建的类对象的__module__特殊属性总是引用types，因此必须手工设置。

综上所述，使用这两种特殊方法定义类要比使用类定义语句定义类麻烦得多。它们的存在更多是为了揭示Python解释器在类定义过程中执行了哪些操作。在绝大部分情况下都没有必要使用这两种特殊方法。

5-3. 类的实例化

（教程：9.3.2、9.3.3、9.3.5）
（语言参考手册：3.2、3.3.1）

不论使用类定义语句、type()还是new_class()，最终得到的都是一个类对象。第3章已经说明，类对象属于可调用对象（因为其元类实现了魔术属性__call__）。但调用类与调用函数是完全不同的两个概念：调用函数的目的是以给定的实际参数执行函数体得到一个返回值；而调用类的目的是以给定的初始化参数获得一个属于该类的对象。这被称为该类的“实例化（instantiation）”，而在这一过程中并不会执行类体。类体在整个脚本执行过程中只会被执行一次，其作用是定义该类本身，而非获得一个该类的“实例（instance）”。

在第3章中已经给出了类的实例化的例子，下面回顾一下：

```
>>> p1 = Point2D(3.2, 0.0)

>>> p2 = Point2D(10, 10)
```

这两条语句其实是对类Point2D的调用，类名之后的圆括号内给出的是“初始化参数列表（initialization argument lists）”。然而在第3章中没有说明这两条语句背后执行的操作，下面我们对其进行讨论。

一个类被调用后，会将该类本身作为第一个参数，与初始化参数列表拼在一起形成一个实际参数列表，以调用该类的魔术属性__new__。第3章已经说明，由于object实现了__new__，所以所有对象其实都具有__new__，但仅对于类对象来说__new__才有意义。严格来说，__new__是一个静态方法（这会在本章后面解释）。__new__的作用就是完成类的实例化。我们可以通过在类定义中包含关于__new__的函数定义语句来覆盖掉默认由object实现的__new__，但这仅会出现在自定义元类的场景中（详见第15章），且绝大多数情况下自定义的__new__都会在函数体内调用默认的__new__。

正常情况下，__new__的返回值是被调用类的一个实例，并执行了如下操作：

➤ 将该实例的特殊属性__class__设置为引用被调用类，以使该实例获得类属性。
➤ 如果被调用类没有__slots__魔术属性，则为该实例添加特殊属性__dict__，并使其引用变量字典（或它的一个只读代理）；否则，为__slots__列出的标识符预留槽位，并使它们都引用NotImplemented。不论哪种情况，该实例都做好了被设置实例属性的准备。
➤ 如果被调用类是一个元类、函数的类型、模块的类型或方法的类型，则还需要为该实例分别添加表3-1～表3-4列出的特殊属性。

接下来，该类对象的__init__魔术属性被调用，以完成对该实例的实例属性的初始化。__init__的实际参数列表是这样形成的：将该实例本身作为第一个参数，再与初始化参数列表拼在一起。虽然object已经实现了__init__，但默认的__init__什么也不做，因此类定义经常会包含关于__init__的函数定义语句，例如在Point2D的类体中就包含如下代码：

```
    #在初始化时设置了实例属性x和y。
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

该段代码使每个属于Point2D的对象都具有实例属性x和y。

__init__的函数体与类体存在如下区别：类体中定义的每个标识符都会自动成为类对象的实例属性（亦即成为该类实例的类属性），而__init__的函数体中定义的标识符在函数返回后不会保留，只有给self参数添加的属性才会成为实例的实例属性。初始化参数列表中的参数是被__init__的函数体使用的，并不一定会被实例的实例属性直接引用。举个例子，下面定义的__init__要求初始化参数列表中有3个参数，前两个作为求和或求积的操作数，第三个用于判断执行求和还是求积：

```
    #在初始化过程中设置了实例属性sum和product。
    def __init__(self, x, y, with_product):
        x = float(x)
        y = float(y)
```

```
        self.sum = x + y

        if with_product:
            self.protuct = x * y
```

在面向对象的术语中，__init__被称为类的“构造器（constructors）”。__init__的返回值必须是None，且习惯上我们不会用return语句显式给出该返回值。此外，__new__调用失败会抛出一个异常，导致脚本的执行进入异常处理流程（见第8章），在这种情况下没有新实例被创建，也就无法调用__init__。

综上所述，类的实例是由__new__和__init__配合创建的：__new__返回新创建的实例，设置其类属性，并为实例属性的设置做好准备；而__init__设置该实例的实例属性。除了可以通过让类对象具有或不具有__slots__来控制其实例是否具有__dict__之外，实例的其余特殊属性都是由Python解释器自动控制的，我们在编写脚本时不需要关心。

相信你已经能推断出，语句“Point2D(3.2, 0.0)”的结果是返回Point2D的一个实例，该实例具有实例属性x和y，分别引用3.2和0.0；类似的，语句“Point2D(10, 10)”的结果是返回Point2D的一个实例，该实例具有实例属性x和y，分别引用10.0和10.0。

与类的实例化相反的过程是减少一个对象的引用数（不论是通过del语句删除标识符还是让引用它的标识符引用别的对象）。第3章已经提到过，当一个对象的引用数变为0时，最终会被垃圾回收机制销毁。而如果一个类对象实现了__del__魔术属性，则该类的实例即将被销毁时（注意不是引用数变为0时）会自动调用它的类属性__del__。在面向对象的术语中，__del__被称为类的“析构器（destructors）”。

__del__只有一个参数self，会被传入实例本身。一般而言，我们会在__del__中做一些记录日志之类的操作。特别的，可以在__del__中通过创建一个该对象的新引用来推迟其被销毁的时间，这被称为“重生（regenerate）”。但这是强烈不推荐的，因为不同的Python解释器对重生的处理不相同，最坏情况下可能导致一个对象永远不能被销毁，直到Python解释器停止运行。

此外，关于__del__必须牢记的是，由于一个对象引用数为0后需要等待一段时间才会被垃圾回收机制销毁，而在这期间Python解释器停止运行的话，那些来不及销毁的对象会直接消失（因为Python解释器的内存空间消失了），这种情况下__del__不会被调用。因此__del__是不可靠的，不能依赖它执行关键性的操作。

至此我们阐明了一个类的实例的生存周期。第3章已经说明，Python中的所有对象都属于某个类，就连类对象也属于自己的元类，因此这就是Python中所有对象的生存周期。在一个对象的生存周期中，我们能对该对象做的唯一操作就是访问该对象的属性。关于如何访问属性已经在第3章详细讨论，下面只补充一些第3章没提到的内容。

属性的访问是通过“.”运算符实现的，其语法在第3章已经给出：

```
object.attribute
```

其中attribute是属性名，可能是实例属性，也可能是类属性，还可能是特殊属性。下面考虑不同范畴的属性发生名字冲突时如何处理。

由于特殊属性在__init__被调用之前就已经设置好了，即使在__init__的函数体内通过self参数设置了与特殊属性同名的“实例属性”，也会退化为对特殊属性的设置，因此特殊属性与实例属性不可能发生名字冲突。但要小心，在__init__的函数体内对实例的特殊属性进行设置常常会带来意想不到的效果，比如下面这个例子：

```
#定义类B。
class B:
    #类属性b。
    b = 0


#定义类A。
class A:
    #类属性a。
    a = 1

    #在初始化时设置__class__特殊属性。
    def __init__(self):
        self.__class__ = B
```

请将上述代码保存为quirk_class_definition.py，然后通过如下命令行和语句来验证关于类A的定义具有非常怪癖的行为：

```
$ python3 -i quirk_class_definition.py

>>> obj = A()
>>> obj.a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'B' object has no attribute 'a'
>>> obj.b
0
>>> obj.__class__
<class '__main__.B'>
>>> isinstance(obj, A)
False
>>> isinstance(obj, B)
True
>>>
```

也就是说，虽然实例obj是通过调用类A创建的，但却不是A的实例，而是B的实例，这是因为类A的__init__在初始化其实例时将实例的特殊属性__class__修改成引用类B。

再进一步，由于一个对象的类属性就是它所属类的实例属性，因此类属性也不可能与特殊属性发生名字冲突。

因此，可能发生名字冲突的情况只可能一个是实例属性另一个是类属性。在这种情况下，一般规则是实例属性覆盖类属性，但引用了数据描述器的属性除外（本章后面会解释）。下面通过如下命令行和语句验证默认情况下实例属性覆盖类属性：

```
$ python3 -i Point2D.py

>>> p0 = Point2D(100, 100)
>>> p0.x
100
>>> Point2D.x = -100
>>> p0.x
100
>>> p0.__class__.x
-100
>>>
```

需要特别强调，访问实例属性时，可以读取、设置和删除；而访问类属性时，只能读取，设置会导致创建一个同名的实例属性，删除则会报错。请通过如下命令行和语句验证：

```
$ python3 -i Point2D.py

>>> p0 = Point2D(1, 2)
>>> Point2D.dimension
2
>>> p0.dimension
2
>>> p0.dimension = 3
>>> Point2D.dimension
2
>>> p0.dimension
3
>>> del p0.dimension
>>> Point2D.dimension
2
>>> p0.dimension
2
>>> del p0.dimension
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: dimension
>>>
```

5-4. 方法、类方法和静态方法

（教程：9.3.4）
（语言参考手册：3.2）
（标准库：内置函数）

从语法上讲，实例属性像类属性一样，既可以是函数属性又可以是非函数属性。但面向对象的基本逻辑是用类把对象分类，然后在类中实现该类的实例能够执行的操作。因此在实际编程时，实例属性几乎总是非函数属性，用于储存仅对该实例有意义的数据（例如Point2D中的x和y）。类属性大部分情况下是函数属性，用于实现该类的实例能执行的操作（例如Point2D中的move和location）；而非函数类属性用于储存对该类的实例都具有意义且取相同值的数据（例如Point2D中的dimension）。

事实上，非函数类属性很多情况下在逻辑上是常量（也就是说Point2D中的dimension按照PEP 8推荐其实应写作DIMENSION），然而Python中没有定义常量的语法，其内置常量和标准库中的其他常量是通过C定义的。（在Python中可以通过反射依靠变量实现常量的效果，这会在下节讨论。）

当然，上述模型仅考虑了普通类的实例化，而忽略了元类的实例化：类对象本身也是元类的实例，但其实例属性大部分是函数属性。

实例属性只能通过实例访问，而类属性既可以通过类对象访问也可以通过实例访问。对于非函数类属性来说，通过类对象访问和通过实例访问在读取时是一样的，而后者不允许设置和删除（这在上一节已经说明）；但对于函数类属性，通过类对象访问相当于一次函数调用，而通过实例访问相当于一次方法调用，两者有明显的区别。

与函数对象和类对象一旦定义就在内存空间中长期存在不同，方法对象是在通过实例访问函数类属性时临时创建的，并在被调用后自动失去所有引用。第3章已经提到了，方法对象是对函数对象的包装，其__func__特殊属性引用被包装的函数，__name__、__doc__和__module__与被包装函数的同名特殊属性引用相同的对象。这意味着方法的名字、文档和所属模块与它所包装的函数的名字、文档和所属模块总是相同的。

方法对象与它包装的函数对象最大的区别在于前者具有__self__特殊属性，该属性引用了访问函数类属性的实例。这意味着方法对象将实例和函数类属性绑定了起来，进而实现了本章开头所说的面向对象的核心理念：将类型与属于该类型的对象可执行的操作绑定起来，以防止因对某对象执行不合适的操作而导致错误。换句话说，当通过实例访问函数类属性时，并不会直接调用该函数类属性，而是调用包装了该函数类属性的方法。

第3章已经说明方法的类型实现了__call__魔术属性，因此方法对象也是可调用对象。一个方法对象被调用时会执行如下操作：将__self__引用的实例作为第一个实际参数，与显式给出的实际参数列表拼接起来，用于调用__func__引用的函数。因此，当进行方法调用时，相关函数类属性的第一个形式参数总是被传入相应实例，而显式给出的实际参数列表会被传入相关函数类属性的后续形式参数。

这就是为什么Point2D类定义中，三个函数类属性__init__、move和location的第一个形式参数都是“self”，它表明需要传入实例本身。而第3章也给出了访问这三个函数类属性的语句，下面回顾一下（实例化隐含对__init__的调用）：

```
>>> p1 = Point2D(3.2, 0.0)
>>> p1.location()
>>> p1.move(3.4, -6.9)
```

注意它们都省略了对应形式参数self的实际参数，而事实上三种情况下传给self的都是标识符p1引用的对象。

当通过类对象访问函数类属性时不涉及方法调用，而是执行普通的函数调用。上面三条语句中的后两条其实可以等价写为：

```
>>> Point2D.location(p1)
>>> Point2D.move(p1, 3.4, -6.9)
```

然而这样做没有利用面向对象的核心优势：当进行方法调用时，只有相关实例所属类中定义了相应函数类属性才会成功，否则会在函数类属性被调用前就抛出异常。请看下面的例子：

```
>>> p2 = object()
>>> p2.location()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'object' object has no attribute 'location'
>>> p2.move(3.4, -6.9)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'object' object has no attribute 'move'
>>>
>>> Point2D.location(p2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Users/wwy/Point2D.py", line 17, in location
    print("("+str(self.x)+", "+str(self.y)+")")
AttributeError: 'object' object has no attribute 'x'
>>> Point2D.move(p2, 3.4, -6.9)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Users/wwy/Point2D.py", line 12, in move
    self.x += dx
AttributeError: 'object' object has no attribute 'x'
>>>
```

可以看出，当通过实例访问函数类属性时，在调用函数类属性之前就已经报错；而当通过类对象访问函数类属性时，要执行到函数类属性的函数体的某条语句时才会报错（甚至可能不会报错）。显然，方法调用比函数调用要安全得多，因此函数类属性几乎总是通过实例访问，我们也可以直接将它们称为“方法”。

然而，并非所有函数类属性都是方法。存在两种例外，即“类方法（class methods）”和“静态方法（static methods）”。类方法是这样一种函数类属性：不论通过实例访问它还是通过类对象访问它，其第一个形式参数都会被传入该类对象（而非实例）。静态方法则是这样一种函数属性：不论通过实例访问它还是通过类对象访问它，其形式参数列表都只会被传入显式给出的实际参数列表。

类方法和静态方法分别通过内置函数classmethod()和staticmethod()实现。这两个内置函数总是被当成装饰器来使用。由于类体中包含的也不过是普通的函数定义语句，因此在类体中可以通过第4章介绍的方法使用装饰器。下面扩展Point2D类，给它增加一个类方法和一个静态方法：

```
class Point2D:
    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)+")")

    #该类方法将非函数类属性dimension重置为默认值。
    @classmethod
    def reset(cls):
        cls.dimension = 2

    #该静态方法输出一段说明文本，并以给定的符号做装饰。
    @staticmethod
    def declare(char='*'):
        n = 17
        line1 = ''
        while n > 0:
            line1 = line1 + char
            n = n - 1
        line2 = char+'                '+char
        print(line1)
        print(line2)
        print(char+' class Point2D '+char)
        print(line2)
        print(line1)
```

请将上述代码保存为Point2D_extended.py，然后通过如下命令行和语句验证类方法和静态方法的用法：

```
$ python3 -i Point2D_extended.py

>>> p0 = Point2D(0, 0)
>>> Point2D.dimension
2
>>> Point2D.dimension = 4
>>> Point2D.dimension
4
>>> Point2D.reset()
>>> Point2D.dimension
2
>>> Point2D.dimension = 8
>>> Point2D.dimension
8
>>> Point2D.reset()
>>> Point2D.dimension
2
>>>
>>> Point2D.declare()
*****************
*               *
* class Point2D *
*               *
*****************
>>> p0.declare('@')
@@@@@@@@@@@@@@@@@
@               @
@ class Point2D @
@               @
@@@@@@@@@@@@@@@@@
>>>
```

最后，类定义中方法和类方法的第一个形式参数可以是其他标识符，但这会降低代码的可读性。PEP 8建议，方法的第一个形式参数总是使用“self”，而类方法的第一个形式参数总是使用“cls”。

5-5. 反射机制

（语言参考手册：3.3.2、3.3.2.1）
（标准库：内置函数）

至此我们已经介绍了在Python中定义类并将其实例化的基本技巧。从本节开始将介绍一些Python面向对象编程中较高级的技巧。本节讨论“反射（reflection）”机制，它不仅让我们更深刻地理解Python中属性访问的底层逻辑，同时也具有相当大的实用价值，可用于实现多种语法特性。

在计算机科学中反射指的是运行中的计算机程序检测和修改其自身状态和行为的能力，最初由布莱恩·史密斯（Brian Smith）于1982年提出，对Java的设计产生了深远的影响。本书不准备深入讨论反射的原理，有兴趣的读者可以参考由布莱恩·富特（Brian Foote）撰写的“Objects, Reflection, and Open Languages”[1]。在本书中，我们可以将反射简单地理解为通过字符串访问一个对象的属性的能力。

第3章已经说明，属性和变量在逻辑上是等同的。而在第6章会讲到，全局变量其实是全局名字空间中的属性。因此Python中的一切对象访问都可以归结为对属性的访问。第3章介绍

的object实现的魔术属性中，__getattribute__、__setattr__和__delattr__与属性访问密切相关，共同实现了反射机制。

当读取一个对象的某属性时，会在该对象所属类的继承链上搜索__getattribute__并执行。具体来说，当以如下形式读取属性时：

```
obj.attribute
```

等价于调用：

```
object.__getattribute__(obj, 'attribute')
```

而object.__getattribute__的默认行为是：依次在obj对象和它所属类的继承链的变量字典中查找“attribute”健，如果找到则返回该键引用的对象，否则抛出AttributeError异常。

当设置一个对象的某属性时，会在该对象所属类的继承链上搜索__setattr__并执行。具体来说，当以如下形式设置属性时：

```
obj.attribute = value
```

等价于调用：

```
object.__setattr__(obj, 'attribute', value)
```

而object.__setattr__的默认行为是：在obj对象的变量字典中查找“attribute”健，如果找到则将其引用的对象设置为value，否则向变量字典中添加“attribute”引用value的键值对。注意该操作总是成功的，且必然作用于实例属性。

当删除一个对象的某属性时，会在该对象所属类的继承链上搜索__delattr__并执行。具体来说，当以如下形式删除属性时：

```
del obj.attribute
```

等价于调用：

```
object.__delattr__(obj, 'attribute')
```

而object.__detattr__的默认行为是：在obj对象的变量字典中查找“attribute”健，如果找到则将其对应的键值对删除，否则抛出AttributeError异常。

通过利用反射机制，内置函数getattr()、setattr()、delattr()和hasattr()使我们能够通过字符串形式的属性名进行属性访问。

getattr()的语法为：

```
getattr(object, name[, default])
```

其行为是：如果object.name存在则返回它；如果object.name不存在，则当default被传入了一个对象时就返回它，否则抛出AttributeError异常。getattr()的等价Python实现为：

```
def getattr(obj, name, default=NotImplemented):
    try:
        return object.__getattribute__(obj, name)
    except AttributeError as err:
        if default is not NotImplemented:
            return default
        else:
            raise err
```

setattr()的语法为：

```
setattr(object, name, value)
```

其行为等价于执行了“object.name = value”。setattr()的等价Python实现为：

```
def setattr(obj, name, value):
    object.__setattr__(obj, name, value)
```

delattr()的语法为：

```
delattr(object, name)
```

其行为等价于执行了“del object.name”。delattr()的等价Python实现为：

```
def delattr(obj, name):
    object.__delattr__(obj, name)
```

最后，hasattr()的语法为：

```
hasattr(object, name)
```

其行为是：如果object.name存在则返回True，否则返回False。事实上，hasattr()是通过检查执行“getattr(object, name)”时是否抛出AttributeError异常来判断指定对象是否具有指定属性的，因此其等价Python实现为：

```
def hasattr(obj, name):
    try:
        getattr(obj, name)
        return True
    except AttributeError:
        return False
```

我们可以通过反射机制实现一些有趣的语法特性，例如前面已经提到了，可以通过反射依靠变量实现常量的效果。下面给出具体的例子：

```
#定义一个代表北欧符文的类。
class Rune:
    #初始化时设置了如下实例属性：
    #  salutation: 符文的称呼，常量。
    #  pronunciation: 符文的发音，常量。
    #  number: 该符文的当前数量，变量。
    def __init__(self, s, p, n):
        self.salutation = s
        self.pronunciation = p
        self.number = n

    #重写__setattr__使得只能修改number属性，且无法添加其他实例属性。
    def __setattr__(self, name, value):
        #只允许设置salutation属性和pronunciation属性一次。
        if name == 'salutation' or name == 'pronunciation':
            if name in self.__dict__:
                return None
        #不允许设置未在__init__中添加的属性。
        elif name != 'number':
            return None
        else:
            pass
        object.__setattr__(self, name, value)

    #重写__delattr__使得无法删除实例属性。
    def __delattr__(self, name):
        return None
```

请将上述代码保存为Rune.py，然后通过如下命令行和语句验证上面的例子通过变量实现了常量：

```
$ python3 -i Rune.py

>>> r1 = Rune('FENU', 'f', 0)
>>> r1.salutation
'FENU'
>>> r1.pronunciation
'f'
>>> r1.number
0
>>> r1.salutation = 'URUZ'
>>> r1.salutation
'FENU'
>>> del r1.salutation
>>> r1.salutation
'FENU'
>>> r1.pronunciation = 'u'
>>> r1.pronunciation
'f'
>>> del r1.pronunciation
>>> r1.pronunciation
'f'
>>> r1.number = 1
>>> r1.number
1
>>> del r1.number
>>> r1.number
1
>>> r1.other_attr = ""
>>> r1.other_attr
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Rune' object has no attribute 'other_attr'
>>>
```

注意虽然类Rune不具有__slots__属性，但依然不能添加没有在__init__中设置的实例属性。这是因为在调用__setattr__时如果属性名不是“salutation”、“pronunciation”或“number”就会直接返回None。而当属性名是“salutation”或“pronunciation”时，如果它们已经在变量字典中，也会直接返回None，这意味着它们只能被设置一次，即只允许在__init__中被设置。当试图删除这些属性时，由于__delattr__会直接返回None，所以无法成功。这就使得Rune类实例的salutation属性和pronunciation属性相当于是常量。

这里需要特别强调，绝对不能在__getattribute__、__setattr__和__delattr__的函数体内以“self.name”的形式访问被操作的属性，因为这会引起这些魔术属性的递归调用。同理，也不能通过内置函数getattr()、setattr()、delattr()和hasattr()来访问被操作的属性。但我们可以将属性名当成键直接操作变量字典，也可以调用object实现的这些魔术属性的默认版本。另外，如果重写这些魔术属性时不在最后调用它们被object类实现的默认版本，就相当于屏蔽了默认行为。

有些时候，我们希望在读取一个对象不具有的属性时返回一个默认值。这理论上也可以通过重写__getattribute__来实现，但__getattribute__本身比较复杂，需要遍历对象的变量字典、它所属类的变量字典和该类的所有基类的变量字典，因此重写起来非常麻烦。为了解决这个问题，Python 3.7引入了魔术属性__getattr__：如果一个对象以类属性的方式具有了__getattr__，则当读取该对象不具有的属性时不会抛出AttributeError异常，而是自动调用__getattr__。下面的例子说明了__getattr__的作用：

```
class Area:
    #该类的实例只具有两个实例属性：长度和高度。
    def __init__(self, l, h):
        self.length = l
        self.height = h

    #添加了__getattr__使得访问该类的实例的任何其他属性时都返回面积，就好像面积是该类
    # 实例的默认属性值。
    def __getattr__(self, name):
        return self.length * self.height
```

请将上述代码保存为Area.py，然后通过如下命令行和语句验证上面的例子实现了属性的默认值：

```
$ python3 -i Area.py

>>> a = Area(10.5, 7.9)
>>> a.length
10.5
>>> a.height
7.9
>>> a.area
82.95
>>> a.volume
82.95
>>> a.color
82.95
>>>
```

5-6. 描述器

（语言参考手册：3.3.2.2、3.3.2.3）
（标准库：functools）

概括地说，当一个类实现了魔术属性__get__、__set__和__delete__中至少一个时，其实例就被称为“描述器（descriptors）”。在实际应用中几乎所有描述器都会实现__get__。如果一个描述器额外实现了__set__和/或__delete__，则该描述器被称为“数据描述器（data descriptor）”。

仅当描述器被某个类属性引用，并通过该类属性被访问时，其特殊性才会体现。我们把具有引用描述器的属性的类称为该描述器的“所有者类（owner class）”。不论通过实例还是类对象读取描述器，描述器的__get__都会被调用，并以其返回值作为读取到的对象。但仅当通过实例设置描述器时，描述器的__set__才会被调用，并将被赋值的对象作为实际参数之一传入。同样，仅当通过实例删除描述器时，描述器的__delete__才会被调用。

我们首先讨论非数据描述器，并将注意力集中到__get__的作用上。__get__被调用时，其第一个参数self总是被传入描述器本身，而instance参数和owner参数被传入的对象遵循如下规则：

➤ 如果描述器是通过实例被访问的，则instance参数被传入该实例，owner参数取默认实参值None。
➤ 如果描述器是通过类对象被访问的，则instance参数被传入None，owner参数被传入该类对象。

当然，在__get__的函数体中是否使用这三个参数由编程者自行决定。

下面给出一个使用非数据描述器的简单例子：

```
from datetime import datetime, timezone, timedelta


#定义Clock类，其实例代表处于某时区的本地时钟。    由于该类实现了__get__，所以该类的实例
# 都是描述器。
class Clock:
    #实例属性tz储存着该时钟所在时区。
    def __init__(self, h):
        h = int(h)
        delta = timedelta(hours=h)
        self.tz = timezone(delta)

    #读取该时钟时返回“hh:mm:ss”格式的本地时间。
    def __get__(self, instance, owner=None):
        t = datetime.now(self.tz)
        return str(t.hour) + ':' + str(t.minute) + ':' + str(t.second)


#定义WorldClock类，其每个类属性以一个城市命名，引用一个处于该城市所在时区的时钟描述器。
class WorldClock:
    honolulu = Clock(-11)    #西11区，火奴鲁鲁。
    hawaii = Clock(-10)    #西10区，夏威夷。
    juneau = Clock(-9)    #西9区，朱诺。
    los_angeles = Clock(-8)    #西8区，洛杉矶。
    mexico_city = Clock(-7)    #西7区，墨西哥城。
    chicago = Clock(-6)    #西6区，芝加哥。
    washington = Clock(-5)    #西5区，华盛顿。
    buenos_aires = Clock(-4)    #西4区，布宜诺斯艾利斯。
    brasilia = Clock(-3)    #西3区，巴西利亚。
    greenland = Clock(-2)    #西2区，格陵兰岛。
    conakry = Clock(-1)    #西1区，科纳克里。
    london = Clock(0)    #UTC时间，伦敦。
    berlin = Clock(1)    #东1区，柏林。
    cairo = Clock(2)    #东2区，开罗。
    moscow = Clock(3)    #东3区，莫斯科。
    abu_dhabi = Clock(4)    #东4区，阿布扎比。
    new_delhi = Clock(5)    #东5区，新德里。
    almaty = Clock(6)    #东6区，阿拉木图。
    singapore = Clock(7)    #东7区，新加坡。
    beijing = Clock(8)    #东8区，北京。
    tokyo = Clock(9)    #东9区，东京。
    melbourne = Clock(10)    #东10区，墨尔本。
    honiara = Clock(11)    #东11区，霍尼亚拉。
    wellington = Clock(12)    #东12区，惠灵顿。
```

该例子从标准库中的datetime模块引入了datetime类、timezone类和timedelta类来处理时间，但本书不会讨论datetime模块。由于Clock类实现了魔术属性__get__，所以它的每个实例都是描述器；而WorldClock类中每个以地名为属性名的类属性都引用了一个Clock类描述器。请将上述代码保存为clock_descriptor.py，然后通过如下命令行和语句验证描述器的特性：

```
$ python3 -i clock_descriptor.py

>>> wc = WorldClock()
>>> wc.london
'13:54:12'
>>> WorldClock.beijing
'21:54:31'
>>> WorldClock.__dict__['cairo']
<__main__.Clock object at 0x101fec6d0>
>>>
```

下面对验证结果进行分析。当执行“wc.london”时，是通过实例wc读取类WorldClock的类属性london引用的描述器，这符合__get__被调用的条件，实际上执行的是如下语句：

```
WorldClock.london.__get__(WorldClock.london, wc)
```

而由于__get__的函数体中并没有使用instance参数，因此被传入的实例wc其实是多余的。该魔术属性会从WorldClock.london引用的Clock描述器的实例属性tz取得该时钟所在时区（也就是UTC时间），然后计算该时区下的当前时间，最后从当前时间中提取出小时数、分钟数和秒数，整理成“hh:mm:ss”格式的字符串返回。而从实例wc的角度看，读取其类属性london得到的就是字符串“13:54:12”，它无法判断该字符串是被london直接引用的，还是由london引用的描述器动态生成的。

当执行“WorldClock.beijing”时，是通过类对象WorldClock读取其类属性beijing引用的描述器，这同样符合__get__被调用的条件，但此时实际上执行的是如下语句：

```
WorldClock.beijing.__get__(WorldClock.beijing, None, WorldClock)
```

而由于__get__的函数体也没有使用owner参数，因此被传入的类对象WorldClock同样是多余的。该魔术属性会以相同的方式从WorldClock.beijing引用的Clock描述器获取该时钟所在时区（东8区），进而计算当前时间，并整理成“hh:mm:ss”格式的字符串返回。

而当执行“WorldClock.__dict__['cairo']”时，是从类对象WorldClock的变量字典中查找键“cairo”对应的值，虽然它与类属性cairo引用的是同一个对象，但这不符合__get__被调用的条件，因此读取到的是描述器本身，即一个属于Clock类的对象。类似的，当一个对象（非类对象）的实例属性引用了描述器，或一个全局标识符引用了描述器，则通过它们读取描述器时也都不符合__get__被调用的条件，会读取到描述器本身。可通过如下语句验证：

```
>>> wc.another_clock = Clock(0)
>>> wc.another_clock
<__main__.Clock object at 0x101fef9d0>
>>> cl = Clock(1)
>>> cl
<__main__.Clock object at 0x101fef940>
>>>
```

上面的简单例子说明了非数据描述器的原理，也给出了它们的基本用法。Python解释器本身广泛使用了非数据描述器来实现各种语法特性，而这些用法要精妙得多。举例来说，方法、类方法和静态方法，其实都是通过非数据描述器实现的。下面依次说明如何通过非数据描述器将类定义中的函数动态转换为方法、类方法和静态方法。

首先需要记住一点，Python中所有的函数都是非数据描述器。请通过如下语句查看函数的类型实现了哪些魔术属性：

```
>>> def f():
...    pass
...
>>> vars(type(f))
mappingproxy({'__new__': <built-in method __new__ of type object at
0x105f41858>, '__repr__': <slot wrapper '__repr__' of 'function' objects>,
'__call__': <slot wrapper '__call__' of 'function' objects>, '__get__': <slot
wrapper '__get__' of 'function' objects>, '__closure__': <member '__closure__' of
'function' objects>, '__doc__': <member '__doc__' of 'function' objects>,
'__globals__': <member '__globals__' of 'function' objects>, '__module__':
<member '__module__' of 'function' objects>, '__builtins__': <member
'__builtins__' of 'function' objects>, '__code__': <attribute '__code__' of
'function' objects>, '__defaults__': <attribute '__defaults__' of 'function'
objects>, '__kwdefaults__': <attribute '__kwdefaults__' of 'function' objects>,
'__annotations__': <attribute '__annotations__' of 'function' objects>,
'__dict__': <attribute '__dict__' of 'function' objects>, '__name__': <attribute
'__name__' of 'function' objects>, '__qualname__': <attribute '__qualname__' of
'function' objects>})
>>>
```

注意函数的类型实现了__get__，但没有实现__set__和__delete__，所以函数是非数据描述器。

函数的类型是以C代码实现的，并内置于Python解释器中，但其__get__的等价Python实现如下：

```
class FunctionType:
    ...

    def __get__(self, instance, owner=None):
        if instance is None:
            return self
        else:
            return MethodType(self, instance)
```

因此对于函数类属性来说，当通过类对象本身调用该属性时，首先要读取该属性引用的函数对象，此时函数的类型的__get__返回该函数对象本身，进而执行函数调用。而当通过属于该类的实例调用该属性时，同样首先要读取该属性引用的函数，但此时函数类型的__get__返回的是将该函数与该实例包装成的方法对象，进而执行方法调用。（MethodType是方法的类型，同样是用C实现的。）

前面已经说明，类方法是通过在类定义中以classmethod()作为函数装饰器得到的，而静态方法是通过在类定义中以staticmethod()作为函数装饰器得到的。但需要说明的是，虽然在第4章已经详细讨论过函数装饰器的原理，并指出任何可调用对象都能作为函数装饰器，但该章给出的例子都是以一个函数作为另一个函数的装饰器。而classmethod()和staticmethod()虽然属于内置函数，但本质上其实都是类。它们是以类作为函数装饰器的例子，因此与以函数作为函数装饰器的情况稍有不同。虽然这两个内置函数都是用C代码实现的，但为了便于讨论，下面会分别给出它们的Python等价实现。

classmethod()的Python等价实现是：

```
class classmethod:
    #创建classmethod类的实例时，将被装饰的目标函数以实例属性的形式记录下来。
    def __init__(self, func):
        self.func = func

    #当以读取描述器的方式访问classmethod类的实例时，可以通过owner参数取得发起读取
    # 操作的类对象，或者通过instance参数取得发起读取操作的实例。   对于后一种情况，
    # 通过其__class__属性取得相关类对象。   最后显式调用目标函数的__get__，并将类
    # 对象传入其self和instance参数。   这最终会得到一个将类对象与目标函数绑定的方法
    # 对象。
    def __get__(self, instance, owner=None):
        if owner is None:
            cls = instance.__class__
        else:
            cls = owner
        return self.func.__get__(cls, cls)
```

而staticmethod()的Python等价实现是：

```
class staticmethod:
    #创建staticmethod类的实例时，将被装饰的目标函数以实例属性的形式记录下来。
    def __init__(self, func):
        self.func = func

    #当以读取描述器的方式访问staticmethod类时，直接返回目标函数，不传入发起读取操作
    #的类对象或实例。    这最终会得到一个函数对象。
    def __get__(self, instance, owner=None):
        return self.func
```

当将它们被用作函数装饰器时，会将目标函数作为初始化参数传入__init__，进而被一个classmethod类的实例或staticmethod类的实例的实例属性func引用，而原类定义中的相应函数类属性其实引用的是该classmethod实例或staticmethod实例。由于这两种实例不是函数对象，不具有特殊属性__name__、__doc__、__annotations__、__module__和__qualname__，所以没有必要使用第4章中介绍的functools.wraps()。而classmethod类和staticmethod类都实现了魔术属性__get__，所以它们的实例本身也是非数据描述器，因此对原类定义中的相应函数类属性的读取会调用__get__，进而分别实现类方法和静态方法的语法特性。

综上所述，classmethod()和staticmethod()的原理本质上是以描述器装饰了描述器，对最终得到对象的读取会导致调用外层描述器的__get__，而内层描述器的__get__是否会被调用，取决于外层描述器的__get__如何实现。在实现类方法时，外层描述器的__get__的函数体内调用了内层描述器的__get__；而在实现静态方法时，外层描述器的__get__的函数体内没有调用内层描述器的__get__。

在有些情况下，当以描述器装饰描述器时，我们希望调用外层描述器的__get__自动转化为调用内层描述器的__get__。这可以通过在外层描述器的__get__中显式调用内层描述器的__get__（类似于classmethod()的__get__）来实现，但比较麻烦。

因此标准库中的functools模块为我们提供了partialmethod()，其语法为：

class **functools.partialmethod(*func*, /, **args*, ***keywords*)**

它与functools.partial()的功能基本相同，区别是func参数一般会被传入一个描述器，而返回的是partialmethod类型的描述器——它的__get__被调用时会自动转换为对被传入的描述器的__get__的调用。因此我们可以functools.partialmethod()的返回值作为描述器的装饰器，以得到上述期待的语法特性。（其实functools.partialmethod()也支持给func参数传入一个非描述器的可调用对象，此时外层描述器的__get__调用会自动转换为对该可调用对象本身的调用。）

至此我们讨论完了非数据描述器，接下来讨论数据描述器。由于数据描述器的应用几乎总是与封装联系在一起，而封装会在下一节详细讨论，所以这里只给出一个简单例子。

在前面已经说明，通过实例访问类属性时只能读取，设置默认会创建一个同名的实例属性，删除默认会报错。下面的例子通过使用数据描述器覆盖了默认行为，使得通过实例设置和删除类属性时会输出一段提示信息：

```
#该类是一个数据描述器。
class Dimension:
    def __init__(self, value):
        self.value = int(value)

    def __get__(self, instance, owner=None):
        return self.value

    #仅当通过实例设置该描述器时会被调用。
    def __set__(self, instance, value):
        print("Instances cannot change the value of its class attributes.")
        return None

    #仅当通过实例删除该描述器时会被调用。
    def __delete__(self, instance):
        print("Instances cannot delete its class attributes.")
        return None


class Point2D:
    #该属性引用一个数据描述器。
    dimension = 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Point2D_data_descriptor.py，然后通过如下命令行和语句验证数据描述器的性质：

```
$ python3 -i Point2D_data_descriptor.py

>>> p = Point2D(0, 0)
>>> p.dimension
2
>>> p.dimension = 3
Instances cannot change the value of its class attributes.
>>> del p.dimension
Instances cannot delete its class attributes.
>>> Point2D.dimension = 3
>>> p.dimension
3
>>> del Point2D.dimension
>>> p.dimension
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
AttributeError: 'Point2D' object has no attribute 'dimension'
>>>
```

上面的例子证明，当通过类对象设置、删除类属性时，__set__和__delete__不会被调用。这导致我们无法通过数据描述器用变量实现常量的效果。

上面的例子还说明了很重要的一点：数据描述器类属性的优先级高于实例属性，也就是说它们不会被同名的实例属性覆盖。注意这与前面介绍的实例属性覆盖类属性的一般规则正好相反。这语法特性使得我们可以通过数据描述器实现只读属性。事实上，上面例子中的dimension属性已经是一个只读属性。

5-7. 封装

（教程：9.4、9.6）
（语言参考手册：3.3.3、6.2.1）
（标准库：内置函数、functools）

学术界总结了面向对象的3大核心特征：

➤ “封装（encapsulation）”：通过对属性的访问控制杜绝因对某对象执行不适合的操作而导致的错误。
➤ “继承（inheritance）”：类之间通过父子关系使一个类自动具有其基类的属性，以减少代码中的冗余。
➤ “多态（polymorphism）”：使一个操作被应用于不同类的实例时，在保持内在逻辑不变的前提下能自动进行微调，以适应不同的类。

由于多态的实现依赖于类型理论，所以被本书列为了高级话题，放到第15章讨论。本节和下一节将分别介绍Python中的封装和继承。

大部分面向对象语言通过将属性分为“公有属性（public attributes）”和“私有属性（private attributes）”来实现封装：一个对象的私有属性只能在它所属类的方法的函数体中通过特殊语法特性（例如this）访问，而在其他情况下都只能访问该对象的公有属性。这些编程语言往往在类定义中用关键字public/private显式将一个属性声明为公有/私有，并通过编译器/解释器确保对私有属性的访问只发生在合法位置。

我们可以认为Python中也有公有属性和私有属性这对概念，但这只是一种约定，而非通过Python解释器强制实现的语法特性。第2章已经提到了如下两种特殊标识符：
_*：以一个下划线开头的标识符，视为私有实例属性或私有类属性。
__*：以两个或更多下划线开头且至多以一个下划线结尾的标识符，视为私有类属性。
而这里的两个“视为”暗示了它们只是一种约定：不要通过实例或类对象访问具有这两种属性名的属性，它们都只应在所属类的方法的函数体中被访问。

也许你会觉得Python实现封装的方式不够严谨，因为对私有属性的保护不是强制性的。但我个人认为，鉴于封装所要解决的问题是相互合作的程序员因沟通不畅而错误理解对方编写

函数的使用方法，并非抵御怀有恶意的攻击者的蓄意破坏，所以Python实现封装的方式是行之有效的——只要所有参与项目的程序员都遵守不直接访问私有属性的约定即可。

私有类属性总是为其他类属性（包括公有类属性和其他私有类属性）提供服务，所以在任何场合都只能在后者的函数体中访问它们。而这又存在两种情况：

➤ 私有类属性只为该类自身的其他类属性提供服务，该类的子类不应访问它们。
➤ 私有类属性为该类和其子类的其他类属性提供服务，这在很多编程语言（例如C++）中被称为“保护属性（protected attributes）”。

第一种私有类属性的属性名应以两个以上下划线开头且至多以一个下划线结尾，解释器在执行脚本时会自动给它们的属性名添加“_ClassName”形式的前缀（其中“ClassName”为定义该类属性的类的类名）。举个例子，如果在类A中定义了类属性“__abc”，则它会被替换为“_A__abc”。这一机制被称为“私有名称转换（private name mangling）”。如果转换后得到的标识符长度超过了255个字符，则会发生截断，但不同Python解释器的截断方法是不同的。而第二种私有类属性的属性名应以一个下划线开头，这样它们就不会被进行私有名称转换。

私有实例属性有可能是为类属性提供服务的，也有可能是因为如下原因被封装的：它们储存着该实例的关键性数据，必须保证这些数据不被破坏。如果是后一种情况，必须通过类属性提供以合法方式访问私有实例属性的接口。

下面的例子给出了经典面向对象技术针对这种情况的解决方案：

```
class Grade:
    #实例属性_value的取值范围是1~5，只能取整数。
    def __init__(self, v):
        v = int(v)
        if v > 5:
            v = 5
        if v < 1:
            v = 1
        self._value = v

    #读取_value属性的方法。
    def getV(self):
        return self._value

    #设置_value属性的方法。   用同__init__的方法确保取值范围是1~5，只能取整数。
    def setV(self, v):
        v = int(v)
        if v > 5:
            v = 5
        if v < 1:
            v = 1
        self._value = v
```

```
        #删除_value属性的方法。   直接返回以禁止删除_value。
        def deleteV(self):
            return
```

请将上述代码保存为encapsulation1.py，然后通过如下命令行和语句验证Python中的封装：

```
$ python3 -i encapsulation1.py

>>> g = Grade(10)
>>> g.getV()
5
>>> g.setV(0)
>>> g.getV()
1
>>> g.deleteV()
>>> g.getV()
1
>>> g._value
1
>>> g._value = -1
>>> g.getV()
-1
>>> del g._value
>>> g.getV()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Users/wwy/encapsulation1.py", line 13, in getV
    return self._value
AttributeError: 'Grade' object has no attribute '_value'
>>>
```

从这个例子可以看出，如果遵守约定，仅通过getV()、setV()和deleteV()来访问_value，则_value引用的对象总是合法的，实现了封装。但如果直接访问_value，则可以破坏其引用对象的合法性，甚至可以删除该属性本身。

上面的例子从逻辑上实现了封装，但它太麻烦了：一个私有实例属性需要三个公有类属性为其服务——读取、设置和删除操作需要分别调用相应的公有类属性。显然，在实际应用中以这样的方式实现封装是不现实的。下面介绍Python中封装的实用解决方案，即所谓的“托管属性（managed attributes）”。

托管属性的原理是：定义一个数据描述器类来描述合法访问方式，然后将所有需要以这种方式访问的私有实例属性实现为相应公有类属性对这种描述器的引用。下面用一个例子加以说明：

```
#此数据描述器用于保证托管属性引用一个浮点数。
class FloatValidator:
    #用该描述器本身的实例属性managed储存被托管属性的名称_name。    而name就是引用该
    # 描述器的公有属性的名称。
    def __set_name__(self, owner, name):
        self.managed = '_' + name

    #当通过实例读取name属性时，转化为读取_name属性。    当通过类对象读取name属性时，
    # 返回描述器本身。    读取到的浮点数保留两位小数。
    def __get__(self, instance, owner=None):
        if instance is None:
            return self
        else:
            value = getattr(instance, self.managed, NotImplemented)
            if value is not NotImplemented:
                value = round(value, 2)
            return value

    #当通过实例设置name属性时，转化为设置_name属性。    设置时要保证赋值浮点数，默认
    # 为0.0。
    def __set__(self, instance, value):
        try:
            value = float(value)
        except ValueError:
            value = 0.0
        setattr(instance, self.managed, value)

    #当通过实例删除name属性时，转化为删除_name属性。
    def __delete__(self, instance):
        delattr(instance, self.managed)


class Person:
    #访问托管属性_height的接口。
    height = FloatValidator()

    #访问托管属性_weight的接口。
    weight = FloatValidator()

    #初始化。
    def __init__(self, h, w):
        self.height = h
        self.weight = w


class Wallet:
    #访问托管属性_coin的接口。
    amount = FloatValidator()

    #初始化。
    def __init__(self):
        self.amount = 0.0
```

请将上述代码保存为encapsulation2.py，然后通过如下命令行和语句验证托管属性的作用：

```
$ python3 -i encapsulation2.py

>>> p = Person(180, 65)
>>> p.height
180.0
>>> p.weight
65.0
>>> p._height
180.0
>>> p._weight
65.0
>>> p.weight = 71.768
>>> p.weight
71.77
>>> p._weight
71.768
>>> del p.weight
>>> p.weight
NotImplemented
>>> p._weight
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Person' object has no attribute '_weight'. Did you mean:
'_height'?
>>> p.weight = 67.1
>>> p.weight
67.1
>>> p._weight
67.1
>>>
>>> w = Wallet()
>>> w.amount
0.0
>>> w.amount = 100.05
>>> w.amount
100.05
>>> Wallet.amount
<__main__.FloatValidator object at 0x10ca381c0>
>>> w.amount = "adfas"
>>> w.amount
0.0
>>>
```

下面对托管属性技术进行分析。

➤ 首先，所有者类的类定义语句中没有在任何地方显式定义被托管的私有属性（例子中的_height、_weight和_coin），而只定义了用于访问这些私有属性的公有属性（例子中的height、weight和coin）。但所有者类的实例都自动具有了被托管的私有属性。这是因为在所有者类的__init__的函数体中会对公有属性进行初始化，使数据描述器的__set__被调用，进而为实例添加相关私有属性。

➤ 其次，数据描述器通过__get__、__set__和__delete__实现对托管属性的访问进行限制。例子中的__set__将一切不能转化为浮点数的对象自动影射到0.0，__get__则自动对读取的浮点数舍入到小数点后两位。而例子中的__delete__使得托管属性（私有属性）可以被

删除，但访问它的接口（公有属性）依然被保留。在这种情况下，再次读取公有属性将得到__get__返回的默认值，而再次设置公有属性将重新添加被托管的私有属性。

➤ 最后，为了让数据描述器具有通用性，其定义中实现了魔术属性__set_name__。该魔术属性仅会在所有者类的定义语句被执行期间被调用，因此在任何数据描述器的生存期中都只会被调用一次。__set_name__会在__init__之后被调用，而在其函数体内，可以通过owner参数取得所有者类，通过name参数取得所有者类中引用该数据描述器的属性的名称，依靠这些信息就能使被托管的私有实例属性的属性名各不相同，进而保证多个同类数据描述器相互间不会干扰。

在上面的例子中，数据描述器之间的独立性是这样实现的：当__set_name__被调用时，以在引用数据描述器的公有属性的属性名前添加“_”的方式生成被托管的私有属性的属性名，然后将生成的属性名储存到描述器自身的实例属性managed中。这意味着托管属性_x和访问它的接口x总是成对出现。在__get__、__set__和__delete__的函数体中，从managed属性取得被托管的私有属性的属性名，然后通过内置函数getattr()、setattr()和delattr()操作被托管的私有属性。前面已经说明了，这些内置函数是依赖于反射实现的。

以上面例子中的方式实现托管属性仅适用于有大量私有实例属性需要以相同的方式访问的情况。很多时候，对某一属性的访问限制只适用于该属性本身，为该属性定义数据描述器类会导致代码过于冗繁。

在这种情况下正确的做法是使用内置函数property()，其语法为：

```
class property(fget=None, fset=None, fdel=None, doc=None)
```

它与classmethod()和staticmethod()类似，本质上是类。下面先给出property()的Python等价实现：

```
class property:

    #初始化时记录下用于设置数据描述器的参数。
    def __init__(self, fget=None, fset=None, fdel=None, doc=None):
        self._fget = fget
        self._fset = fset
        self._fdel = fdel
        #当没有显式给出数据描述器的文档时，优先尝试使用fget的文档。
        if doc is None and fget is not None:
            doc = fget.__doc__
        self.__doc__ = doc

    #初始化时记录下托管属性的属性名。
```

```
        def __set_name__(self, owner, name):
            self._managed = '_' + name

        #当读取该数据描述器时，调用fget。
        def __get__(self, instance, owner=None):
            if instance is None:
                return self
            else:
                if self._fget is None:
                    msg = 'attribute ' + self._managed + ' is unreadble'
                    raise AttributeError(msg)
                else:
                    return self._fget(instance)

        #当通过实例设置该数据描述器时，调用fset。
        def __set__(self, instance, value):
            if self._fset is None:
                msg = 'attribute ' + self._managed + ' is unwritable'
                raise AttributeError(msg)
            else:
                return self._fset(instance, value)

        #当通过实例删除该数据描述器时，调用fdel。
        def __delete__(self, instance):
            if self._fdel is None:
                msg = 'cannot delete attribute ' + self._managed
                raise AttributeError(msg)
            else:
                return self._fdel(instance)

        #返回用于更新fget的装饰器。
        def getter(self, fget):
            prop = type(self)(fget, self._fset, self._fdel, self.__doc__)
            prop._managed = self._managed
            return prop

        #返回用于更新fset的装饰器。
        def setter(self, fset):
            prop = type(self)(self._fget, fset, self._fdel, self.__doc__)
            prop._managed = self._managed
            return prop

        #返回用于更新fdel的装饰器。
        def deleter(self, fdel):
            prop = type(self)(self._fget, self._fset, fdel, self.__doc__)
            prop._managed = self._managed
            return prop
```

请对比property()的类定义和前一个例子中关于数据描述器的类定义。

property()最原始的用法，是先分别定义需要传入参数fget、fset和fdel的函数，再调用一次property()直接得到所需装饰器。注意传入fget和fdel的函数只需要一个用于接收实例的参数，而传入fset的函数则额外需要一个用于接收值的参数。下面的例子说明了这种用法：

```
class Wallet:
    #在初始化时设置被托管的私有属性。
    def __init__(self):
        self._amount = 0.0

    #传入fget的函数。
    def get_amount(self):
        if self._amount is not NotImplemented:
            return round(self._amount, 2)
        else:
            return self._amount

    #传入fset的函数。
    def set_amount(self, value):
        try:
            value = float(value)
        except ValueError:
            value = 0.0
        self._amount = value

    #传入fdel的函数。
    def del_amount(self):
        self._amount = NotImplemented

    #作为访问私有属性接口的公有属性。
    amount = property(get_amount, set_amount, del_amount)
```

请将上述代码保存为encapsulation3.py，然后通过如下命令行和语句验证这同样实现了托管属性（与前一个例子的效果一模一样）：

```
$ python3 -i encapsulation3.py

>>> w = Wallet()
>>> w.amount
0.0
>>> w.amount = 37.487234
>>> w.amount
37.49
>>> w.amount = "adfas"
>>> w.amount
0.0
>>> del w.amount
>>> w.amount
NotImplemented
>>> w.amount = 20
>>> w.amount
20.0
>>>
```

但要注意以这种方式封装同样需要为每个私有实例属性提供三个公有类属性，因此并没有达到优化代码的目的。

在实际应用中，我们会将property()本身，以及property()实例化出的数据描述器的getter属性、setter属性和deleter属性当成装饰器来使用，进而优化代码。下面的例子是对上面例子的改写，请将其保存为encapsulation4.py：

```
class Wallet:
    def __init__(self):
        self._amount = 0.0

    #在定义作为访问私有属性接口的公有属性的同时实现了fget。
    @property
    def amount(self):
        if self._amount is not NotImplemented:
            return round(self._amount, 2)
        else:
            return self._amount

    #更新fset。
    @amount.setter
    def amount(self, value):
        try:
            value = float(value)
        except ValueError:
            value = 0.0
        self._amount = value

    #更新fdel。
    @amount.deleter
    def amount(self):
        self._amount = NotImplemented
```

可以验证，这样实现封装的效果与前一个例子完全相同，但只需要为被托管的私有属性提供一个公有属性作为接口。具体来说，上面的例子在定义数据描述器amount时以property()作为装饰器，使其具有可用的fget；然后以数据描述器本身的setter属性作为装饰器，使其具有了可用的fset；最后以数据描述器本身的deleter属性作为装饰器，使其具有了可用的fdel。这一顺序不是固定的，可以任意更改，例如先通过property()给amount添加fset，再通过其deleter属性添加fdel，最后通过其getter属性添加fget。

此外请注意，property()作为装饰器时是以类作为装饰器，而property()实例化出的对象的getter、setter和deleter属性作为装饰器时是以方法作为装饰器，这再次扩展了作为装饰器的可调用对象的范围。

这里值得一提的是，第3章曾经说明如果一个类具有__slots__，则会自动为它所指定的每个标识符创建一个数据描述器。事实上，该数据描述器与encapsulation4.py中的数据描述器在逻辑上是相同的。

最后，标准库中的functools模块定义的cached_property()函数实现了一种特殊意义上的封装：将一个基于其他实例属性动态生成的值缓存起来，当成一个虚拟的实例属性来使用。

该函数的语法为：

@functools.cached_property(*func*)

我们几乎总是将它当成装饰器来使用。

下面用一个例子来说明cached_property()的用法：

```
import time
import functools


class CachedFibonacci:
    #以实例属性储存斐波拉契序列的索引。
    def __init__(self, i):
        self.index = i

    #斐波拉契序列生成器，私有类属性。
    def __fibonacci(self, n):
        if n < 2:
            return 1
        else:
            return self.__fibonacci(n-1) + self.__fibonacci(n-2)

    #基于实例属性动态生成斐波拉契序列的元素。    这虽然是一个类属性，但通过不同实例访
    # 问得到不同的值。
    @functools.cached_property
    def element(self):
        return self.__fibonacci(self.index)


#定义一个统计访问CachedFibonacci实例的element属性所需时间的函数。
def process_time(cf):
    starttime = time.time()
    print('element: ' + str(cf.element))
    endtime = time.time()
    print('time: ' + str(endtime - starttime))
```

请将上述代码保存为encapsulation5.py，然后通过如下命令行和语句验证这种封装方式的基本特征：

```
$ python3 -i encapsulation5.py

>>> cf1 = CachedFibonacci(30)
>>> cf2 = CachedFibonacci(20)
>>> process_time(cf1)
element: 1346269
time: 0.3900718688964844
>>> process_time(cf2)
element: 10946
time: 0.004217863082885742
>>>
```

这说明虽然对于cf1和cf2来说element是类属性，但该属性引用的对象是基于实例属性index动态生成的，所以读取的结果各不相同。

接下来执行如下语句：

```
>>> process_time(cf1)
element: 1346269
time: 3.3855438232421875e-05
>>> process_time(cf2)
element: 10946
time: 3.314018249511719e-05
>>>
```

这说明由于element属性被cached_property()装饰，所以在第一次通过实例读取时会执行其函数体，而后续的读取将直接从缓存中取得结果，不再执行函数体。

接下来执行如下语句：

```
>>> cf1.index = 10
>>> process_time(cf1)
element: 1346269
time: 2.4080276489257812e-05
>>> del cf1.element
>>> process_time(cf1)
element: 89
time: 6.985664367675781e-05
>>>
```

这说明即便动态生成element属性引用的对象时所基于的index属性引用的对象发生了变化，element属性引用的对象并不会随之改变。换句话说，缓存中的对象不会自动失效，必须通过删除element属性来清空缓存。

接下来执行如下语句：

```
>>> cf2.element = 0
>>> process_time(cf2)
element: 0
time: 2.5987625122070312e-05
>>> del cf2.element
>>> process_time(cf2)
element: 10946
time: 0.003716707229614258
>>>
```

这说明设置element属性会导致创建一个同名的实例属性，将虚拟实例属性覆盖；而之后删除element属性不仅会删除实例属性element，还会清空类属性element的缓存。

最后请执行如下语句：

```
>>> CachedFibonacci.element
<functools.cached_property object at 0x109b6bd90>
>>>
```

这说明通过类对象读取element属性会得到经cached_property()装饰的函数本身。

5-8. 继承

（教程：9、9.5、9.6）
（语言参考手册：3.3.1）
（标准库：内置函数）

我们已经知道Python中的类支持多继承，而访问一个对象的类属性时需要在其所属类的继承链上基于属性名搜索，然而到目前为止本书并没有描述该搜索的具体细节。本章开头介绍了定义类的三种方式，但并没有给出涉及继承的例子。本节将弥补这些缺失的内容。

为了便于讨论，我们先给出一个多继承的例子。请执行如下语句：

```
>>> class X:
...     pass
...
>>> class Y:
...     pass
...
>>> class Z:
...     pass
...
>>> class A(X, Y):
...     pass
...
>>> class B(Z):
...     pass
...
>>> class C(A, B):
...     pass
...
>>>
```

通过这些语句定义的多个类相互间具有如图5-1所示的继承关系。

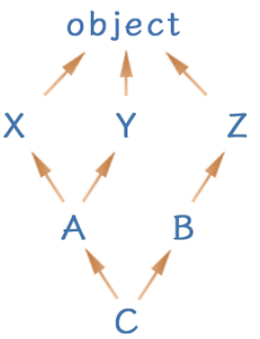

图5-1. 继承例子1

那么访问一个属于类C的对象的类属性时将按照什么顺序搜索类C的继承链呢？在第3章中已经提到过，类对象的__mro__特殊属性引用一个以类对象为成员的元组，其中类对象的排列顺序与该类的基类被继承的顺序相同。现在需要进一步说明，这一顺序也就是在继承链中搜索类属性的顺序，而“MRO”其实是“方法解析顺序（method resolution order）”的缩写（由于大部分类属性都是方法，所以这里的“方法”代表类属性）。请执行如下语句：

```
     >>> C.__mro__
     (<class '__main__.C'>, <class '__main__.A'>, <class '__main__.X'>, <class
'__main__.Y'>, <class '__main__.B'>, <class '__main__.Z'>, <class 'object'>)
     >>>
```

可以推断出在类C的继承链中搜索类属性的顺序是：C → A → X → Y → B → Z → object。

不论用本章开头介绍的哪种方法定义类，都会计算该类的MRO，并将其储存在相应类对象的__mro__特殊属性中。此后在该类的继承链中搜索类属性时，直接使用__mro__储存的计算结果即可。所以__mro__引用的对象总是自动生成的，不能手工设置。

用于计算MRO的算法被称为“C3方法（C3 method）”，可以被描述为：

```
L[object] = object
L[C(B1 ... BN)] = C + merge(L[B1], ... , L[BN], B1 ... BN)
```

其中L是个算子，代表计算指定类的MRO，结果用一个类序列表示。类后面的括号中的类序列代表它的直接基类。merge是另一个算子，表示合并多个类序列。“+”表示将左边的类添加到右边的类序列的头部。

C3方法是一个递归算法，其核心在于合并多个类序列的merge算子，而合并的步骤为：

➤ 步骤一：将结果设置为一个空序列，记为R。
➤ 步骤二：检查当前是否还有某个类序列不为空。如果所有类序列都为空，则跳到步骤五。
➤ 步骤三：将所有类序列都分解为头部和尾部两部分，头部仅包含第一个类，尾部则包含所有剩余的类（只有一个类的类序列的尾部为空）。
➤ 步骤四：从第一个类序列开始，依次检查这些类序列的头部。如果发现某头部没有出现在其余所有类序列的尾部，则将其包含的类添加到R中，然后从所有类序列中删除该类，回到步骤二。如果检查完了所有类序列，没有一个头部可以添加到R中，则跳到步骤六。
➤ 步骤五：合并完成，结果是R。
➤ 步骤六：合并失败，拒绝定义指定的类，并抛出TypeError异常。

下面的等式说明了生成上面例子中类C的__mro__引用的对象的计算过程（O代表object）：

```
      L[C] = C + merge(L[A], L[B], AB)
           = C + merge(A + merge(L[X], L[Y], XY), B + merge(L[Z], Z), AB)
           = C + merge(A + merge(X + merge(L[O], O), Y + merge(L[O], O), XY), B +
merge(Z + merge(L[O], O), Z), AB)
           = C + merge(A + merge(X + merge(O, O), Y + merge(O, O), XY), B +
merge(Z + merge(O, O), Z), AB)
           = C + merge(A + merge(XO, YO, XY), B + merge(ZO, Z), AB)
           = C + merge(AXYO, BZO, AB)
           = C + AXYBZO
           = CAXYBZO
```

这与前面通过__mro__验证到的结果一致。需要注意的是，在计算merge(XO, YO, XY)时，在将X放入R后，下一个放入R的类是Y而不是O，因为O出现在了YO的尾部，所以最后的结果是XYO而非XOY，这就是merge算子的关键点。

与深度优先搜索算法可以用肉眼观察得到结果不同，C3算法相当复杂。为什么Python要用一个这么复杂的算法来计算MRO呢？事实上，Python 2最初使用的就是深度优先算法，对于上面例子中的C来说，MRO将是C → A → X → object → Y → object → B → Z → object。这一解决方案有两个缺点：

➤ 每次搜索类属性都需要额外开销：如果你对深度优先算法很熟悉，那可以推知用它计算出的MRO中，继承链的每个菱形结构的上顶点都会被搜索多次，次数恰好等于该顶点的入度（即指向该顶点的箭头的数量）。

➤ 重写规则不符合直觉：即便继承链的某个菱形结构的右顶点重写了其上顶点的某个类属性，菱形结构的下顶点依然会访问上顶点的该类属性，使得重写失败。例如在上面的例子中，如果类B重写了object的__lt__，那么类B的实例将使用重写后的__lt__，而类C的实例依然会使用object的__lt__。

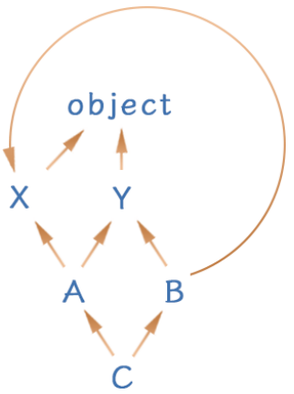

图5-2. 继承例子2

为了解决上述问题，Python 2.2对深度优先搜索算法进行了改进，即如果一个类在MRO中出现了多次，则仅保留最后一次。对上面的例子来说，使用这个改进后的算法将得到与使用C3方法相同的结果。然而请看图5-2给出的例子，按照改进后的算法，C的MRO是C → A → B → Y → X → object，而A的MRO是A → X → Y → object，两者中X和Y的顺序正好相反，因此不能匹配。

C3方法是Python 2.3引入的，确保了MRO是“单调的（monotonic）”：如果在一个类的MRO中类X在类Y之前，则在该类的任何子类的MRO中类X依然在类Y之前。然而这意味着并非任何形式的多继承都能用C3方法计算出MRO。对于上面的例子使用C3方法，A的MRO是A → X → Y → object，B的MRO是B → Y → X → object，两者是矛盾的，所以C的MRO无法得出。下面的等式说明了C3算法将在哪一步报错：

```
     L[C] = C + merge(L[A], L[B], AB)
          = C + merge(A + merge(L[X], L[Y], XY), B + merge(L[Y], L[X], YX), AB)
          = C + merge(A + merge(X + merge(L[O], O), Y + merge(L[O], O), XY), B +
merge(Y + merge(L[O], O), X + merge(L[O], O), YX), AB)
          = C + merge(A + merge(XO, YO, XY), B + merge(YO, XO, YX), AB)
          = C + merge(AXYO, BYXO, AB)
```

而在合并AXYO，BYXO和AB时，首先将A和B依次放入R中，而剩下的merge(XYO, YXO)将无法继续，因为X在YXO的尾部中，Y在XYO的尾部中。

我们可以通过下面的语句验证无法定义上面例子中的类C：

```
>>> class X:
...     pass
...
>>> class Y:
...     pass
...
>>> class A(X, Y):
...     pass
...
>>> class B(Y, X):
...     pass
...
>>> class C(A, B):
...     pass
...
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Cannot create a consistent method resolution
order (MRO) for bases X, Y
>>>
```

综上所述，Python虽然支持多继承，但基类并不能任意组合，需要保证能够通过C3方法计算出单调的MRO。这表面上看是个缺点，但其实是一个优点。如果多继承编程语言没有单调性限制（例如Python 2.2及之前的版本），基类中又广泛存在同名的类属性，那么访问类属性时具体定位到哪个就很难确定。关于C3方法更详细的描述见米歇尔·西米奥纳多（Michele Simionato）撰写的“The Python 2.3 Method Resolution Order”[2]。

MRO本质上是将（可能）具有菱形结构的继承链线性化，因此必然损失很多信息，也就是说无法从MRO本身判断出所涉及的类相互之间的继承关系。举例来说，C → A → X → Y → B → Z → object可能是图5-1中的继承链产生的，也可能由按照其顺序依次单继承的继承链产生。

为了判断一个类是否是另一个类的子类，应使用内置函数issubclass()，其语法为：

```
issubclass(class, classinfo)
```

其中class参数需传入一个类，而classinfo参数可以被传入一个类，一个类形成的元组，或一个类的联合类型（从Python 3.10开始支持，见第15章）。如果该函数返回True，则说明前者是后者（中的某个类）的子类。

类似的，为了判断一个对象是否是一个类的实例，应使用内置函数isinstance()，其语法为：

```
isinstance(object, classinfo)
```

其中object参数需传入一个对象，而classinfo参数与issubclass()中相同。需要强调的是，本书之前通过比较一个对象的__class__引用的类与另一个类是否相等来判断该对象是否是后者的实例，isinstance()与这种方法并不等同，因为如果A是B的基类，obj是属于B的对象，则obj.__class__不等于A，但isinstance(obj, B)会返回True。

当访问对象的类属性时，会按照MRO搜索其所属类的继承链，并访问第一个匹配者。因此MRO中位于前面的类的属性可以屏蔽位于后面的类的同名属性，这就是重写机制的原理。重写对于非函数类属性、方法、类方法和静态方法都是有效的。然而由于属性名以两个或更多下划线开头且至多以一个下划线结尾的类属性会被进行私有名称转换，所以它们其实具有全

局唯一性（全局名字空间中不存在同名的类），无法通过重写屏蔽。我们必须遵守这样的约定：即便在其子类的类属性的函数体中，也不要访问该类会被进行私有名称转换的类属性。

如果B的直接基类中只有A具有__slots__，且B自身未定义__slots__，则B会继承A的__slots__，使得B的实例同时具有通过__dict__引用的变量字典和引用对应__slots__的C数组的槽位。这可以通过如下语句证明：

```
>>> class A:
...     __slots__ = ('a', 'b')
...
>>> class B(A):
...     pass
...
>>> B.__slots__
('a', 'b')
>>> obj = B()
>>> obj.a = 0
>>> obj.b = 1
>>> obj.c = 2
>>> obj.__dict__
{'c': 2}
>>>
```

如果B的直接基类中只有A具有__slots__，但B自身也定义了__slots__，则B的__slots__会屏蔽A的__slots__，B的实例只具有引用C数组的槽位，然而该C数组对应的是两个__slots__指定的标识符的并集。这可以通过如下语句证明：

```
>>> class A:
...     __slots__ = ('a', 'b')
...
>>> class B(A):
...     __slots__ = ('b', 'c')
...
>>> B.__slots__
('b', 'c')
>>> obj = B()
>>> obj.a = 0
>>> obj.b = 1
>>> obj.c = 2
>>>
```

注意该规则是可递归的，也就是说如果C的直接基类中只有B具有__slots__，而C自身也定义了__slots__，则C的__slots__会屏蔽B的__slots__，然而C的实例可以具有的实例属性是三个__slots__指定的标识符的并集。

如果一个类的直接基类中有两个或更多具有__slots__（不论是它们自身定义的还是继承而来的），则在定义该类时会抛出TypeError异常。此外，如果一个类派生自某种内置类型，则不应具有__slots__。

在很多情况下需要特意去访问某个被屏蔽掉的类属性，此时需要使用内置函数super()，其语法为：

```
class super([type[, object-or-type]])
```

该内置函数其实也是一个类，被调用时会创建一个用于访问指定类属性的特殊代理对象，两个参数则用于生成一个临时MRO：

➤ object-or-type：用于指定被截取的MRO，如果是一个对象则取该对象所属类的MRO，如果是一个类则取该类本身的MRO。
➤ type：用于指定从哪个位置开始截取，必须是一个在MRO上的类，而截取的结果是从该类后面一个类开始直到MRO的最后。

super()返回的代理对象属于super类型，变量字典是空的，但却可以通过它访问到生成的临时MRO中每个类的属性（但只支持obj.attr这种方式，不支持抽取和切片）。

当通过super()访问方法时，object-or-type会被传入该方法的self参数，因此只能是一个对象。当通过super()访问类方法时，类方法的cls参数会被传入该类方法所属类，object-or-type无限制。当通过super()访问静态方法时，object-or-type必须是一个类（虽然它不会被作为静态方法的实际参数）。

以C → A → X → Y → B → Z → object为例，假设obj是属于类C的对象，则super(C, obj)和super(C, C)等价，返回的代理对象对应的临时MRO为A → X → Y → B → Z → object，包含了C的所有基类；而super(Y, obj)和super(Y, C)等价，返回的代理对象对应的临时MRO为B → Z → object。

super()最常见的出现位置是在定义类属性的函数的函数体中，在这种情况下通常会省略它的两个参数。当通过属于该类或该类的子类的对象访问这样的类属性时，super()的type参数是该类，而object-or-type参数是该对象。下面用一个例子证明这点：

```
class A:
    def who(self):
        return 'A'

class B(A):
    def who(self):
        return 'B'

    def parent(self):
        print ('My parent is ' + super().who())

class C(B):
    def who(self):
        return 'C'

class D(A):
```

```
        def who(self):
            return 'D'

    class E(C, D):
        def who(self):
            return 'E'
```

通过这些语句定义的多个类相互间具有如图5-3所示的继承关系。

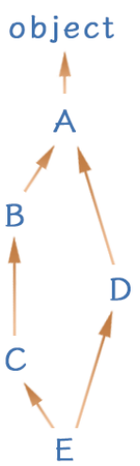

图5-3. 继承例子3

请将上述代码保存为super.py，然后通过下面的命令行和语句验证若调用super()时省略了实参列表，将按照前面的描述传入实际参数：

```
$ python3 -i super.py

>>> b = B()
>>> b.parent()
My parent is A
>>> c = C()
>>> c.parent()
My parent is A
>>> e = E()
>>> e.parent()
My parent is D
>>>
```

下面对该例子进行解释。上面定义的所有类都具有类属性who，但只有B具有类属性parent。不论通过属于B的对象还是属于B的子类（C、E）的对象访问parent，在其函数体内调用super()都等价于调用super(B, self)。B的MRO为B → A → object，因此super()返回的代理对象的MRO为A → object，导致访问A的类属性who。C的MRO为C → B → A → object，因此super()返回的代理对象的MRO也是A → object，同样导致访问A的类属性who。而E的MRO为E → C → B → D → A → object，因此super()返回的代理对象的MRO是D → A → object，导致访问D的类属性who，尽管D事实上只是B的兄弟类而非基类。

MRO对于类属性的继承非常重要，但对于实例属性的继承却毫无影响。那么实例属性的继承又是怎么实现的呢？这就要回到通过__new__、__init__和__del__实现的对象生存周

期。概括地说，当需要考虑继承时，重写这三个魔术属性都需要在最后通过super()显式调用其基类的相应魔术属性。

我们极少会重写__new__，而是直接使用由object实现的__new__。如果确实要重写__new__，则它有固定的重写格式：

```
@staticmethod
def __new__(cls, *args, **kwargs):
    ... you can do anything here ...
    return super().__new__(cls, *args, **kwargs)
```

函数体内对super().__new__的调用确保了object实现的__new__最终会被调用，进而为新建实例分配内存空间。若省略了该语句，虽然不会报错，但包含该重写__new__的类将无法被实例化。此外，虽然__new__的第一个形式参数是cls，但它是一个静态方法，cls必须显式传入被实例化的类。除了cls参数之外，Python解释器在调用__new__时还会传入一些内部参数，且不同解释器可能存在不同，因此必须将通过args和kwargs接收到的参数原封不动地传给基类的__new__。

由于__init__是负责为新建实例设置实例属性的，所以如果A是B的基类，则B的__init__必须要能够接受传给A的__init__的实际参数。故__init__也有明确的重写格式：

```
def __init__(self, ..., *args, ..., **kwargs):
    ... you can do anything here ...
    super().__init__(*args, **kwargs)
```

但与__new__的固定重写格式不同，每个__init__会“消耗掉”的基于位置对应的实际参数和基于关键字对应的实际参数是各不相同的。

需要强调的是，上面的重写格式只适用于单继承，这种情况下一个类可以确定它的任意子类的MRO都有如下性质：子类MRO从该类开始直到末尾的部分总是等于该类自己的MRO。这一性质使我们在进行类的实例化时能够使用基于位置对应的实际参数。然而图5-3对应的例子已经说明，在存在多继承的情况下，一个类无法预知其子类的MRO（例如E → C → B → D → A → object）会给它自身的MRO（例如C → B → A → object）在什么位置插入哪些类。在这种情况下进行类的实例化时只能使用基于关键字对应的实际参数，而__init__的重写格式也需要变成：

```
def __init__(self, ..., **kwargs):
    ... you can do anything here ...
    super().__init__(**kwargs)
```

本书之前涉及类的例子因为确定不会被其他类继承，所以没有遵守上述规范。如果需要支持单继承，那么第3章中定义的Poin2D的__init__可以写为：

```
        def __init__(self, x, y, *args, **kwargs):
            self.x = x
            self.y = y
            super().__init__(*args, **kwargs)
```

相应的创建Point2D的实例时只能写为：

```
    >>> p1 = Point2D(3.2, 0.0)
```

而Point2D的__init__也可以写为：

```
        def __init__(self, *args,  x, y, **kwargs):
            self.x = x
            self.y = y
            super().__init__(*args, **kwargs)
```

此时创建Point2D类的实例只能写为：

```
    >>> p1 = Point2D(x=3.2, y=0.0)
```

如果需要支持多继承，那么Point2D类的__init__只能写为：

```
        def __init__(self, x, y, **kwargs):
            self.x = x
            self.y = y
            super().__init__(**kwargs)
```

而创建Point2D类的实例只能写为：

```
    >>> p1 = Point2D(x=3.2, y=0.0)
```

综上所述，如果你想使一个类可以被任意继承，就应该遵循如下3条建议：
1. 如果该类不需要给其实例设置实例属性，就不要重写__init__。这样该类的子类在被实例化时不会调用该类的__init__（因为不存在），而是直接在该类的基类中搜索__init__。
2. 如果该类重写了__init__，则使用上面的第二种重写格式（即不使用*args）。
3. 在对该类进行实例化时只使用基于关键字对应的实际参数。

object没有实现__del__，因此如果一个类的全部基类（也就是MRO中的所有类）都没有实现__del__，那么该类也可以不实现__del__。但反过来，只要一个类的任何一个基类实现了__del__，那么该类就必须按照如下格式实现__del__：

```
def __del__(self):
    ... you can do anything here ...
    super().__del__()
```

但需要强调，如果一个类的全部基类都没有实现__del__，但它自己选择实现__del__，那么该__del__中不能包含“super().__del__()”语句，否则会抛出AttributeError异常。

最后，本节用一个较真实的例子来总结上面介绍的知识，并给你一个关于继承的直观印象。

假设你在制作一款冒险类游戏，游戏人物使用的武器是这样设计其分类的：所有武器都有物理伤害；火焰武器额外有火焰伤害，冰冻武器额外有冰冻伤害，闪电武器额外有闪电伤害；而魔法武器则同时具有火焰、冰冻和闪电伤害。这四种伤害都有一个最小值和一个最大值，每次攻击的伤害值在这个区间内随机变动。

那么，一个设计良好的继承链将如图5-4所示：

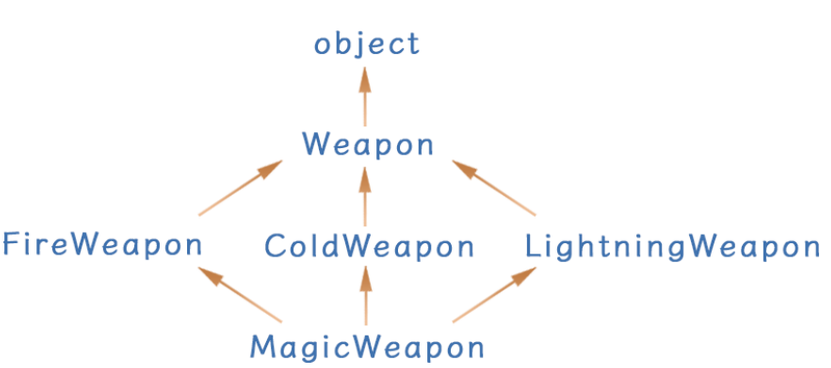

图5-4. 继承例子4

下面给出实现上述继承关系的类定义的代码：

```
from datetime import datetime, timedelta
from random import randint

#定义代表通用武器的类Weapon。    所有其他武器类都以该类为基类。
class Weapon:
    #初始化时设置了如下私有实例属性：
    #   _damage_min: 物理伤害的最小值。
    #   _damage_max: 物理伤害的最大值。
    #   _category: 该武器的类别，例如“大马士革刀”。
    #   _name: 默认引用空串，表明这是普通武器。    如果引用其他字符串，则表明该武器
    # 是有名字的武器。
```

```
        def __init__(self, dmin=0, dmax=0, ctg='Void Blade', name='',
**kwargs):
            self._damage_min = int(dmin)
            self._damage_max = int(dmax)
            self._category = ctg
            self._name = name
            super().__init__(**kwargs)

        #该保护类属性为最近一次掷骰子的时间。
        _time = datetime.now()

        #该私有类属性为一个取值0~100的骰子，对应物理伤害。
        __dice = 0

        #该保护类属性用于掷骰子。
        @classmethod
        def _roll_dice(cls):
            cls.__dice = randint(0, 100)
            cls._time = datetime.now()

        #该保护类属性基于骰子点数计算物理伤害。
        def _physical_damage(self):
            return (self._damage_min * (100-self.__dice) \
                    + self._damage_max * self.__dice) // 100

        #该公有类属性用于计算该武器某次攻击时产生的伤害。
        def damage(self):
            #如果最近一次掷骰子距离现在的时间超过了一秒，则重新掷骰子。
            time_interval = datetime.now() - self._time
            if time_interval > timedelta(seconds=1):
                self._roll_dice()
            #返回伤害值。
            return self._physical_damage()

        #该公有类属性返回该武器的描述。
        def description(self):
            if self._name:
                s = self._name + "'s " + self._category + "\n"
            else:
                s = self._category + "\n"
            s = s + "physical damage: " + str(self._damage_min) + "-"\
                    + str(self._damage_max) + "\n"
            print(s)


    #定义代表火焰武器的类FireWeapon。
    class FireWeapon(Weapon):
        #初始化时设置了如下私有实例属性:
        #  _fire_min: 火焰伤害的最小值。
        #  _fire_max: 火焰伤害的最大值。
        def __init__(self, fmin=0, fmax=0, **kwargs):
            self._fire_min = fmin
            self._fire_max = fmax
            super().__init__(**kwargs)

        #该私有类属性为一个取值0~100的骰子，对应火焰伤害。
        __dice = 0

        #该保护类属性用于掷骰子。
        @classmethod
        def _roll_dice(cls):
            cls.__dice = randint(0, 100)
            super()._roll_dice()

        #该保护类属性基于骰子点数计算火焰伤害。
```

```
        def _fire_damage(self):
            return (self._fire_min * (100-self.__dice) \
                    + self._fire_max * self.__dice) // 100

        #该公有类属性用于计算该武器某次攻击时产生的伤害。
        def damage(self):
            #如果最近一次掷骰子距离现在的时间超过了一秒，则重新掷骰子。
            time_interval = datetime.now() - self._time
            if time_interval > timedelta(seconds=1):
                self._roll_dice()
            #返回伤害值。
            return self._physical_damage() + self._fire_damage()

        #该公有类属性返回该武器的描述。
        def description(self):
            if self._name:
                s = self._name + "'s " + self._category + "\n"
            else:
                s = self._category + "\n"
            s = s + "physical damage: " + str(self._damage_min) + "-"\
                    + str(self._damage_max) + "\n"
            s = s + "fire damage: " + str(self._fire_min) + "-"\
                    + str(self._fire_max) + "\n"
            print(s)


    #定义代表冰冻武器的类ColdWeapon。
    class ColdWeapon(Weapon):
        #初始化时设置了如下私有实例属性:
        #  _cold_min: 冰冻伤害的最小值。
        #  _cold_max: 冰冻伤害的最大值。
        def __init__(self, cmin=0, cmax=0, **kwargs):
            self._cold_min = cmin
            self._cold_max = cmax
            super().__init__(**kwargs)

        #该私有类属性为一个取值0~100的骰子，对应冰冻伤害。
        __dice = 0

        #该保护类属性用于掷骰子。
        @classmethod
        def _roll_dice(cls):
            cls.__dice = randint(0, 100)
            super()._roll_dice()

        #该保护类属性基于骰子点数计算冰冻伤害。
        def _cold_damage(self):
            return (self._cold_min * (100-self.__dice) \
                    + self._cold_max * self.__dice) // 100

        #该公有类属性用于计算该武器某次攻击时产生的伤害。
        def damage(self):
            #如果最近一次掷骰子距离现在的时间超过了一秒，则重新掷骰子。
            time_interval = datetime.now() - self._time
            if time_interval > timedelta(seconds=1):
                self._roll_dice()
            #返回伤害值。
            return self._physical_damage() + self._cold_damage()

        #该公有类属性返回该武器的描述。
        def description(self):
            if self._name:
                s = self._name + "'s " + self._category + "\n"
            else:
                s = self._category + "\n"
```

```
            s = s + "physical damage: " + str(self._damage_min) + "-"\
                    + str(self._damage_max) + "\n"
            s = s + "cold damage: " + str(self._cold_min) + "-"\
                    + str(self._cold_max) + "\n"
            print(s)


    #定义代表闪电武器的类LightningWeapon。
    class LightningWeapon(Weapon):
        #初始化时设置了如下私有实例属性:
        #  _lightning_min: 闪电伤害的最小值。
        #  _lightning_max: 闪电伤害的最大值。
        def __init__(self, lmin=0, lmax=0, **kwargs):
            self._lightning_min = lmin
            self._lightning_max = lmax
            super().__init__(**kwargs)

        #该私有类属性为一个取值0~100的骰子，对应闪电伤害。
        __dice = 0

        #该保护类属性用于掷骰子。
        @classmethod
        def _roll_dice(cls):
            cls.__dice = randint(0, 100)
            super()._roll_dice()

        #该保护类属性基于骰子点数计算闪电伤害。
        def _lightning_damage(self):
            return (self._lightning_min * (100-self.__dice) \
                    + self._lightning_max * self.__dice) // 100

        #该公有类属性用于计算该武器某次攻击时产生的伤害。
        def damage(self):
            #如果最近一次掷骰子距离现在的时间超过了一秒，则重新掷骰子。
            time_interval = datetime.now() - self._time
            if time_interval > timedelta(seconds=1):
                self._roll_dice()
            #返回伤害值。
            return self._physical_damage() + self._lightning_damage()

        #该公有类属性返回该武器的描述。
        def description(self):
            if self._name:
                s = self._name + "'s " + self._category + "\n"
            else:
                s = self._category + "\n"
            s = s + "physical damage: " + str(self._damage_min) + "-"\
                    + str(self._damage_max) + "\n"
            s = s + "lightning damage: " + str(self._lightning_min) + "-"\
                    + str(self._lightning_max) + "\n"
            print(s)


    #定义代表魔法武器的类MagicWeapon。
    class MagicWeapon(FireWeapon, ColdWeapon, LightningWeapon):
        #初始化时没有设置任何私有实例属性:
        def __init__(self, **kwargs):
            super().__init__(**kwargs)

        #该保护类属性用于掷骰子。
        @classmethod
        def _roll_dice(cls):
            super()._roll_dice()

        #该公有类属性用于计算该武器某次攻击时产生的伤害。
```

```
        def damage(self):
            #如果最近一次掷骰子距离现在的时间超过了一秒，则重新掷骰子。
            time_interval = datetime.now() - self._time
            if time_interval > timedelta(seconds=1):
                self._roll_dice()
            #返回伤害值。
            return self._physical_damage() + self._fire_damage()\
                    + self._cold_damage() + self._lightning_damage()

        #该公有类属性返回该武器的描述。
        def description(self):
            if self._name:
                s = self._name + "'s " + self._category + "\n"
            else:
                s = self._category + "\n"
            s = s + "physical damage: " + str(self._damage_min) + "-"\
                    + str(self._damage_max) + "\n"
            s = s + "fire damage: " + str(self._fire_min) + "-"\
                    + str(self._fire_max) + "\n"
            s = s + "cold damage: " + str(self._cold_min) + "-"\
                    + str(self._cold_max) + "\n"
            s = s + "lightning damage: " + str(self._lightning_min) + "-"\
                    + str(self._lightning_max) + "\n"
            print(s)
```

请将上述代码保存到weapons.py中。由于这是本书中代码最多的一个例子，所以下面对它逐步验证和分析。

首先考虑继承链中最顶层（除了object）的基类Weapon。请通过如下命令行和语句查看其实例具有哪些属性：

```
    $ python3 -i weapons.py

    >>> w1 = Weapon()
    >>> vars(w1)
    {'_damage_min': 0, '_damage_max': 0, '_category': 'Void Blade', '_name':
''}
    >>> vars(type(w1))
    mappingproxy({'__module__': '__main__', '__init__': <function
Weapon.__init__ at 0x10f1dfeb0>, '_time': datetime.datetime(2022, 2, 21, 18, 53,
57, 642788), '_Weapon__dice': 0, '_roll_dice': <classmethod(<function
Weapon._roll_dice at 0x10f1dff40>)>, '_physical_damage': <function
Weapon._physical_damage at 0x10f1e0040>, 'damage': <function Weapon.damage at
0x10f1e00d0>, 'description': <function Weapon.description at 0x10f1e0160>,
'__dict__': <attribute '__dict__' of 'Weapon' objects>, '__weakref__': <attribute
'__weakref__' of 'Weapon' objects>, '__doc__': None})
    >>>
```

结合前面的代码与上述结果可知，Weapon类型的实例具有4个私有实例属性：

_category：引用一个字符串，表明该武器的类型。

_name：引用一个字符串，表明该武器的名称。该属性引用空串表明该武器是没有名称的普通武器。

_damage_min：该武器的物理伤害下限。

_damage_max：该武器的物理伤害上限。

由于这些私有实例属性都是为类属性提供服务的，所以没有必要提供访问它们的接口。

除了__init__外，Weapon类型的实例具有4个私有类属性：
_Weapon__dice：一个取值0～100的骰子，用于计算某次攻击的物理伤害。
_roll_dice：代表掷骰子这一动作，是一个类方法。
_time：最近一次掷骰子的时间。
_physical_damage：基于当前骰子的点数计算具体物理伤害。
注意_Weapon__dice是代码中的__dice经私有名称转换得到的，因此是纯粹的私有属性，不能在Weapon的子类中访问；其余三个私有属性则是所谓的“保护属性”，可以在Weapon的子类中访问。

我们其实只能通过如下2个公有类属性来使用Weapon的实例：
damage：用于取得该武器一次攻击造成的伤害。
description：描述该武器，即给出该武器的名称、类型和物理伤害上下限。
damage方法的设计逻辑是这样的：根据人物的攻击速度，武器在一秒钟内可能多次攻击，但每次都掷骰子的计算代价太高（因为要生成一个随机数），所以通过将当前时间与_time记录的时间做比较，将掷骰子的频率限制在每秒最多一次。注意Weapon的所有实例是共用一个骰子的，因此该机制可以极大降低服务器的计算压力。

Weapon的__init__其实只需要4个参数（没算self），分别用于给4个私有实例属性赋值。kwargs参数和函数体中对super()的调用是为了符合前面提到的可任意继承类的__init__的编写规范，其实它总是导致调用object实现的__init__，进而什么也不做。当让Weapon类的__init__属性的所有参数都取默认实参值时，将产生一把攻击力为0的“虚空之刃”，如上面的例子所示。下面使用这把虚空之刃：

```
>>> w1.description()
Void Blade
physical damage: 0-0

>>> w1.damage()
0
>>> w1.damage()
0
...
>>>
```

可以看出，虚空之刃的每一次攻击造成的伤害都是0。

下面的语句创建了一把没有名字的普通匕首：

```
>>> w2 = Weapon(dmin=3, dmax=9, ctg='Dagger')
>>> w2.description()
Dagger
physical damage: 3-9

>>> w2.damage()
8
>>> w2.damage()
3
>>> w2.damage()
```

```
7
...
>>>
```

而下面的语句创建了一把有名字的武器——“赵子龙的长枪”：

```
>>> w3 = Weapon(dmin=97, dmax=168, ctg='Pike', name='Zilong Zhao')
>>> w3.description()
Zilong Zhao's Pike
physical damage: 97-168

>>> w3.damage()
150
>>> w3.damage()
127
>>> w3.damage()
120
...
>>>
```

在理解了Weapon的设计之后，让我们将注意力移到继承链的下一层，即Weapon的直接子类。这包括FireWeapon、ColdWeapon和LightningWeapon。我们需要重点关注它们增加了哪些属性，重写了哪些属性。

首先考察FireWeapon。它通过重写__init__为其实例添加了如下2个私有实例属性：
_fire_min：该武器的火焰伤害下限。
_fire_max：该武器的火焰伤害上限。
而函数体内对super()的调用则会导致调用Weapon的__init__，因此FireWeapon实例也会具有Weapon实例的4个私有实例属性。

FireWeapon还添加了如下两个私有类属性：
_FireWeapon__dice：由代码中的__dice经私有名称转换得到，也是纯粹的私有属性，用于计算某次攻击的火焰伤害。
_fire_damage：基于当前骰子的点数计算具体火焰伤害。
私有类属性_roll_dice则被重写，其函数体内对super()的调用会导致Weapon的_roll_dice被调用。此外，私有类属性_time和_physical_damage则被直接继承。

注意_fire_damage方法使用的是_FireWeapon__dice引用的骰子，而_damage方法使用的是_Weapon__dice引用的骰子，所以两者完全独立互不影响，这也是两个引用骰子的属性应被设置为纯粹的私有属性的原因。然而_roll_dice方法需要在函数体内调用基类的同名属性，以确保一次攻击能够同时触发两个骰子被掷出，所以该属性必须被设置为保护属性。从逻辑上讲，FireWeapon具有两个骰子，一个是自己实现的，一个是基类Weapon实现的。

如果_roll_dice被设置为纯粹的私有属性，即__roll_dice，那么经过私有名称转换后，它将变成_Weapon__roll_dice。当然，理论上我们也可以让super()返回的代理对象直接调用_Weapon__roll_dice，这在语法上和功能上都是正确的，但这意味着如果将来Weapon类被改名，该调用就会失败。所以不在代码中使用经私有名称转换得到的属性名是必须遵守的Python编程规范。

FireWeapon重写了两个公有类属性：
damage：虽然逻辑未变，但返回的伤害是由物理伤害加上火焰伤害得到的。
description：对该武器的描述增加了火焰伤害上下限。

通过考察ColdWeapon和LightningWeapon，可以知道它们的设计逻辑与FireWeapon是一样的，只不过ColdWeapon增加了私有实例属性_cold_min和_cold_max，以及私有类属性_ColdWeapon__dice和_cold_damage；而LightningWeapon则增加了私有实例属性_lightning_min和_lightning_max，以及私有类属性_LightningWeapon__dice和_lightning_damage。

下面的语句创建了一把火焰武器——“红宝石权杖”：

```
      >>> w4 = FireWeapon(dmin=26, dmax=42, fmin=10, fmax=30, ctg='Mace',
name='Ruby')
      >>> w4.description()
      Ruby's Mace
      physical damage: 26-42
      fire damage: 10-30

      >>> vars(w4)
      {'_fire_min': 10, '_fire_max': 30, '_damage_min': 26, '_damage_max': 42,
'_category': 'Mace', '_name': 'Ruby'}
      >>> vars(type(w4))
      mappingproxy({'__module__': '__main__', '__init__': <function
FireWeapon.__init__ at 0x10f1e01f0>, '_FireWeapon__dice': 15, '_roll_dice':
<classmethod(<function FireWeapon._roll_dice at 0x10f1e0280>)>, '_fire_damage':
<function FireWeapon._fire_damage at 0x10f1e0310>, 'damage': <function
FireWeapon.damage at 0x10f1e03a0>, 'description': <function
FireWeapon.description at 0x10f1e0430>, '__doc__': None, '_Weapon__dice': 91,
'_time': datetime.datetime(2022, 2, 21, 20, 10, 42, 788336)})
      >>> w4.damage()
      65
      >>> w4.damage()
      60
      >>> w4.damage()
      53
      ...
      >>>
```

下面的语句创建了一把冰冻武器——“霜之哀伤剑”：

```
      >>> w5 = ColdWeapon(dmin=105, dmax=200, cmin=66, cmax=99, ctg='Sword',
name='Frost Mourn')
      >>> vars(w5)
      {'_cold_min': 66, '_cold_max': 99, '_damage_min': 105, '_damage_max': 200,
'_category': 'Sword', '_name': 'Frost Mourn'}
      >>> vars(type(w5))
      mappingproxy({'__module__': '__main__', '__init__': <function
ColdWeapon.__init__ at 0x10f1e04c0>, '_ColdWeapon__dice': 0, '_roll_dice':
<classmethod(<function ColdWeapon._roll_dice at 0x10f1e0550>)>, '_cold_damage':
<function ColdWeapon._cold_damage at 0x10f1e05e0>, 'damage': <function
ColdWeapon.damage at 0x10f1e0670>, 'description': <function
ColdWeapon.description at 0x10f1e0700>, '__doc__': None})
```

```
>>> w5.description()
Frost Mourn's Sword
physical damage: 105-200
cold damage: 66-99

>>> w5.damage()
207
>>> w5.damage()
224
>>> w5.damage()
187
...
>>>
```

而下面的语句则创建了一把闪电武器——“雷神之锤”：

```
>>> w6 = LightningWeapon(dmin=3000, dmax=6000, lmax=10000, ctg='Hammer',
name='Thor')
>>> vars(w6)
{'_lightning_min': 0, '_lightning_max': 10000, '_damage_min': 3000,
'_damage_max': 6000, '_category': 'Hammer', '_name': 'Thor'}
>>> vars(type(w6))
mappingproxy({'__module__': '__main__', '__init__': <function
LightningWeapon.__init__ at 0x10f1e0790>, '_LightningWeapon__dice': 0,
'_roll_dice': <classmethod(<function LightningWeapon._roll_dice at
0x10f1e0820>)>, '_lightning_damage': <function LightningWeapon._lightning_damage
at 0x10f1e08b0>, 'damage': <function LightningWeapon.damage at 0x10f1e0940>,
'description': <function LightningWeapon.description at 0x10f1e09d0>, '__doc__':
None})
>>> w6.description()
Thor's Hammer
physical damage: 3000-6000
lightning damage: 0-10000

>>> w6.damage()
7010
>>> w6.damage()
13240
>>> w6.damage()
8830
>>>
```

最后，让我们把注意力放到继承链的最低层，即MagicWeapon。它继承了FireWeapon、ColdWeapon和LightningWeapon。

MagicWeapon重写了__init__，但没有添加任何私有实例属性，只是在函数体内通过super()调用了基类的__init__。MagicWeapon的MRO是MagicWeapon → FireWeapon → ColdWeapon → LightningWeapon → Weapon → object，所以其__init__中的super()调用将导致FireWeapon的__init__被调用。然而由于上述类的__init__都通过super()调用了其基类的__init__，所以MRO中的每个类的__init__都会被调用，因此MagicWeapon的实例将具有上面提到的所有私有实例属性。

MagicWeapon重写了_roll_dice，但由于它没有引入新的骰子，所以只在函数体中通过super()调用了基类的_roll_dice。同样，这会导致MRO中除object之外的类的_roll_dice被

依次调用，使得魔法武器的一次攻击会能够同时触发四个骰子被掷出。因此MagicWeapon虽然没有直接继承实现骰子的四个纯粹的私有类属性，却在逻辑上具有四个骰子。

MagicWeapon重写了两个公有类属性：
damage：逻辑仍然未变，但返回的伤害是物理伤害、火焰伤害、冰冻伤害和闪电伤害之和。
description：对该武器的描述需包括四种伤害的上下限。

下面的语句则创建了一把魔法武器——“变色龙的虚空之刃”：

```
      >>> w7 = MagicWeapon(dmin=100, dmax=100, fmin=15, fmax=50, cmin=26,
cmax=32, lmax=99, name="Chameleon")
      >>> type(w7).__mro__
      (<class '__main__.MagicWeapon'>, <class '__main__.FireWeapon'>, <class
'__main__.ColdWeapon'>, <class '__main__.LightningWeapon'>, <class
'__main__.Weapon'>, <class 'object'>)
      >>> vars(w7)
      {'_fire_min': 15, '_fire_max': 50, '_cold_min': 26, '_cold_max': 32,
'_lightning_min': 0, '_lightning_max': 99, '_damage_min': 100, '_damage_max':
100, '_category': 'Void Blade', '_name': 'Chameleon'}
      >>> vars(type(w7))
      mappingproxy({'__module__': '__main__', '__init__': <function
MagicWeapon.__init__ at 0x10f1df760>, '_roll_dice': <classmethod(<function
MagicWeapon._roll_dice at 0x10f1df6d0>)>, 'damage': <function MagicWeapon.damage
at 0x10f1df640>, 'description': <function MagicWeapon.description at
0x10f1df5b0>, '__doc__': None})
      >>> w7.description()
      Chameleon's Void Blade
      physical damage: 100-100
      fire damage: 15-50
      cold damage: 26-32
      lightning damage: 0-99

      >>> w7.damage()
      197
      >>> w7.damage()
      197
      >>> w7.damage()
      232
      ...
      >>>
```

上面的例子就讨论到这里。综上所述，有效使用继承的关键是保持清晰的逻辑，以达到缩减代码的效果。但要强调的是，如果继承链设计的逻辑不清晰，则通常会带来巨大的麻烦，例如很多难以发现的bug。

5-9. 类装饰器

（语言参考手册：3.3.6、8.8）
（标准库：functools）

到目前为止，我们已经见到了以函数、类和方法作为装饰器的例子，但在这些例子中被装饰的对象都是函数，因此没有脱离“函数装饰器”的范畴。本节介绍的“类装饰器”指的是将类作为被装饰的对象，而装饰器本身依然可以是任何可调用对象。

从之前函数装饰器的例子可以看出，当以函数或方法作为装饰器时，最后得到的是其返回值；而当以类作为装饰器时，最后得到的是该类的实例。这一规律对于类装饰器同样适用，只不过返回值/实例包装的是类对象而非函数对象。

下面是一个函数作为类装饰器的例子：

```
#定义被用作装饰器的函数。
def classname(cls):
    if not hasattr(cls, 'classname'):
        def classname(self):
            return type(self).__name__
        classname.__qualname__ = cls.__qualname__ + '.classname'
        classname.__module__ = cls.__module__
        cls.classname = classname
    return cls


#定义被装饰的目标类A。
@classname
class A:
    pass


#定义被装饰的目标B。
@classname
class B:
    pass


#定义被装饰的目标C。
@classname
class C:
    def classname(self):
        return 'Class Name: C'
```

请将上述代码保存到class_decorator1.py中，然后通过如下命令行和语句验证：

```
$ python3 -i class_decorator1.py

>>> a = A()
>>> a.classname()
'A'
>>> a.classname.__qualname__
'A.classname'
>>> a.classname.__module__
'__main__'
>>> b = B()
```

```
>>> b.classname()
'B'
>>> b.classname.__qualname__
'B.classname'
>>> b.classname.__module__
'__main__'
>>> c = C()
>>> c.classname()
'Class Name: C'
>>> c.classname.__qualname__
'C.classname'
>>> c.classname.__module__
'__main__'
>>>
```

可以看出，上面例子中的类装饰器classname的作用是给目标类添加classname属性，当通过目标类的实例访问该属性时会得到目标类的类名。如果目标类已经有classname属性，则该类装饰器什么也不做。

对比class_decorator1.py和function_decorator3.py可知，定义用作类装饰器的函数比定义用作函数装饰器的函数要简单，无需返回一个闭包，也不必使用functools.wraps，只需要修改通过cls参数传入的类对象，然后将其返回即可。需要注意的一点是如果为目标类添加了函数属性，就需要手工设置该属性的__qualname__和__module__，以使其与目标类的其他函数属性保持一致。

下面给出一个类作为类装饰器的例子，它其实与上面例子中的类装饰器功能完全相同，但实现方法却要麻烦一些：

```
#定义被用作装饰器的类。
class classname:
    def __init__(self, cls):
        if not hasattr(cls, 'classname'):
            def classname(self):
                return type(self).__name__
            classname.__qualname__ = cls.__qualname__ + '.classname'
            classname.__module__ = cls.__module__
            cls.classname = classname
        self.cls = cls
        self.__name__ = cls.__name__
        self.__qualname__ = cls.__qualname__
        self.__module__ = cls.__module__
        self.__doc__ = cls.__doc__
        self.__annotations__ = cls.__annotations__
        self.__bases__ = cls.__bases__
        self.__mro__ = cls.__mro__

    def __call__(self, *args, **kwargs):
        return self.cls(*args, **kwargs)


#定义被装饰的目标类A。
@classname
class A:
```

```
        pass


    #定义被装饰的目标B。
    @classname
    class B:
        pass


    #定义被装饰的目标C。
    @classname
    class C:
        def classname(self):
            return 'Class Name: C'
```

请将上述代码保存到class_decorator2.py中，然后通过如下命令行和语句验证：

```
$ python3 -i class_decorator2.py

>>> a = A()
>>> a.classname()
'A'
>>> b = B()
>>> b.classname()
'B'
>>> c = C()
>>> c.classname()
'Class Name: C'
>>> A.__name__
'A'
>>> A.__qualname__
'A'
>>> A.__module__
'__main__'
>>> A.__doc__
>>> A.__annotations__
{}
>>> A.__bases__
(<class 'object'>,)
>>> A.__mro__
(<class '__main__.A'>, <class 'object'>)
>>>
```

可以看出，用类作为类装饰器时，需要在__init__中完成目标类的修饰，并将目标类本身和它的所有特殊属性记录到装饰器实例的实例属性中；然后还需要实现__call__以使装饰器实例可被调用，而调用结果是转为对目标类的调用。（注意这是本书第一个自定义__call__的例子。）简言之，必须将装饰器实例伪装成一个类对象。

事实上，用类作为函数装饰器时，在正常情况下也需要按照上面例子给出的方式将其实例伪装成函数对象。然而当前面讨论描述器时，在涉及用类作为函数装饰器的例子中（即内置函数classmethod()和staticmethod()），由于装饰器的实例是作为类属性被访问的，导致描述器的__get__被调用，所以没有必要实现__call__，也没有必要用实例属性记录目标函数的特殊属性（只需要记录目标函数本身）。

下面给出一个例子说明用类作为函数装饰器的标准方法，它其实是对第4章中关于函数装饰器的例子function_decorator3.py的改写：

```
#定义被用作装饰器的类。
class mydecorator:
    def __init__(self, func):
        self.func = func
        self.__name__ = func.__name__
        self.__qualname__ = func.__qualname__
        self.__module__ = func.__module__
        self.__doc__ = func.__doc__
        self.__annotations__ = func.__annotations__
        self.__code__ = func.__code__
        self.__defaults__ = func.__defaults__
        self.__kwdefaults__ = func.__kwdefaults__
        self.__globals__ = func.__globals__
        self.__closure__ = func.__closure__

    def __call__(self, *args, **kwargs):
        print("I can do anything before calling the target!")
        result = self.func(*args, **kwargs)
        print("I can do anything after calling the target!")
        return result


#定义目标函数1，同时完成装饰。
@mydecorator
def mytarget1():
    print("I am doing nothing.")
    return 0


#定义目标函数2，同时完成装饰。
@mydecorator
def mytarget2(obj):
    print("I am doing the truth value testing.")
    return bool(obj)


#定义目标函数3，同时完成装饰。
@mydecorator
def mytarget3(x, y):
    print("I am an adder.")
    return float(x) + float(y)


#定义目标函数4，同时完成装饰。
@mydecorator
def mytarget4(*substr):
    print("I am an concatenator.")
    length = len(substr)
    if length == 0:
        return ''
    elif length == 1:
        return substr[0]
    else:
        i = 0
        s = ''
```

```
            while i < length:
                s = s + substr[i]
                i = i + 1
            return s
```

请将上面的代码保存为function_decorator5.py，然后通过下面的命令行和语句验证被装饰后的目标函数的行为与在function_decorator3.py中相同：

```
$ python3 -i function_decorator5.py

>>> mytarget1()
I can do anything before calling the target!
I am doing nothing.
I can do anything after calling the target!
0
>>> mytarget2(None)
I can do anything before calling the target!
I am doing the truth value testing.
I can do anything after calling the target!
False
>>> mytarget3(3, 4.5)
I can do anything before calling the target!
I am an adder.
I can do anything after calling the target!
7.5
>>> mytarget4('a', 'b', 'c', 'd')
I can do anything before calling the target!
I am an concatenator.
I can do anything after calling the target!
'abcd'
>>>
```

最后值得在这里一提，标准库中的functools模块提供了total_ordering类装饰器，即：

```
@functools.total_ordering
```

该类装饰器具有这样的作用：只要目标类实现了__eq__，以及__lt__、__le__、__gt__和__ge__中至少一个，那么该装饰器就会自动实现__ne__，以及__lt__、__le__、__gt__和__ge__中没被实现的部分。total_ordering的原理很简单：

1. 有了=，基于not可以构造出!=。
2. 有了=和!=，>和>=中只要有一个就可以基于and构造出另一个，同理<和<=中只要有一个就可以构造出另一个。
3. 有了>，基于not可以构造出<=；有了<，基于not可以构造出>=。

但实现total_ordering的代码却很复杂，因为需要考虑的情况太多，本书就不给出其Python等价实现了。

下面用一个例子说明total_ordering的用法：

```
import functools

#定义一个代表扑克牌的类。
@functools.total_ordering
class Pocker:
    #添加了如下实例属性:
    #  suit: 扑克牌的花色，用1~4分别表示黑桃、红桃、梅花和方块。
    #  rank: 扑克牌的位阶，用1~13分别表示A、2~10和J、Q、K。
    def __init__(self, suit, rank):
        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

    #显示一张扑克的花色和位阶。
    def show(self):
        #识别花色。
        if self.suit == 1:
            s = 'Spade'
        elif self.suit == 2:
            s = 'Heart'
        elif self.suit == 3:
            s = 'Club'
        else:
            s = 'Diamond'
        #识别位阶。
        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)
        #输出结果。
        print(s)

    #两张扑克相等的规则是花色和位阶都相同。
    def __eq__(self, other):
        if self.suit == other.suit and self.rank == other.rank:
            return True
        else:
            return False

    #扑克排序的规则是先比较位阶再比较花色。
    def __lt__(self, other):
        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
```

请将上面的代码保存为Pocker.py，然后执行下面的命令行和语句：

```
$ python3 -i Pocker.py

>>> card1 = Pocker(4, 1)
>>> card1. show()
Diamond A
>>> card2 = Pocker(3, 12)
>>> card2.show()
Club Q
>>> card1 != card2
True
>>> card1 > card2
True
>>> card3 = Pocker(2, 12)
>>> card3.show()
Heart Q
>>> card3 <= card2
False
>>>
```

在上面的例子中，Pocker类只实现了__eq__和__lt__，但被total_ordering装饰后也能正确执行!=、>、<=和>=运算。但必须要强调的是，使用total_ordering的代价是比较运算的执行速度会下降，所以如果追求速度的话最好实现全部6个比较相关魔术属性。

---

[1] Brian Foote. "Objects, Reflection, and Open Languages". www.laputan.org/reflection/openlang.html.

[2] Michele Simionato. "The Python 2.3 Method Resolution Order". https://www.python.org/download/releases/2.3/mro/.