Fluent Python 笔记

Pionpill [1]

本文档为作者学习《Fluent Python》[2]一书时的笔记。

2021 年 12 月 8 日

[1]笔名：北岸，电子邮件：673486387@qq.com，Github：https://github.com/Pionpill

[2]《Fluent Python》:Luciano Ramalho 2017 年中文第一版

前言：

笔者为软件工程系在校本科生，主要利用 Python 进行数据科学与机器学习使用，也有一定游戏开发经验。

«Fluent Python» 是 Python 学习的进阶书籍，在学习本书之前，本人已拜读过 «Python 从入门到实践 »[1]，«Automate with Python» 等书。本书更多聚焦于 Python 内部处理，本人在阅读本书时获得了极大的收获，惊喜程度不亚于初读蟒蛇书。

书上有大量的示例，在本文中也有给出，脚本位于 Scripts 文件下，但本人只书写了部分脚本，还有小部分本人觉得没有必要，或者是以命令行形式书写，这些并没有给出。已有的脚本推荐读者使用 VSCode 和 Python Preview 插件，便于查看脚本运行时的内部逻辑。此外，原文有大段对 Python3[2] 和 Python2 的对比，除非特别必要，本人不会再详述 Python2 [3]的相关内容，除非特殊说明，默认适用于 Python3 和 CPython 解释器。

本笔记不能代替原书，仅是对原书的一个总结归纳；原书大段精妙的解释均没有被记录在笔记中。本人极其推荐有一定 Python 基础的人购买原书阅读，直到截稿日期，我都认为本书是我在学习 Python 路线上阅读过最好的书籍之一。

本笔记只是对原书的马虎概括与整理，如有疑问或需求，还请购买原书；本文所在 Github 仓库采用 GPL v3 协议，但请勿将本文商业使用，本文引用了一些 CSDN 或其他论讨的文章，如果原作者觉得不合适，请联系本人。

2021 年 12 月 8 日

[1]俗称：蟒蛇书

[2]原书支持到 Python3.4

[3]特指 Python2.7 ，再远古的版本不再提及

目录

I 序幕

1 Python 数据模型

1.1 内置函数

1.1.1 内置函数的概念

Python 语言往往使用 len(object) 的方式获取对象的长度，而 C++，Java 等语言在获取类长度时，需要自行在类中定义相关函数，而后采用成员方法或调用成员变量 (object.len()) 的方式获取相关值。

这是由于 python 编译器使用了名为内置函数的方式，这样有许多好处，例如内置函数的计算速度比成员函数快许多，许多常用的操作在定义内置函数后将变得极其容易使用。

内置函数的通常使用 __function__ 的形式。

```
# 例1-1: 几个内置函数的理解
import collections

Card = collections.namedtuple('Card', ['rank', 'suit'])

class FrenchDeck:
    ranks = [str(n) for n in range(2, 11)] + list('JQKA')
    suits = 'spades diamonds clubs hearts'.split()

    def __init__(self):
        self._cards = [Card(rank, suit)
                       for suit in self.suits for rank in self.ranks]

    def __len__(self): # 内置函数 len()
        return len(self._cards)

    def __getitem__(self, position): # 内置迭代器
        return self._cards[position]
```

上例中使用了三个内置函数，其中 __init__ 类似于 C++ 的构造函数，用于创建对象。__len__ 函数的定义使得我们可以调用 len() 函数，而 __getitem__ 的定义允许我们对 FrenchDeck 类创建的对象进行迭代，以及类似 C++ 中的数组操作[1]。

[1]原文对这些操作做了精巧的解释，如果读者不清楚，还请阅读原文

1.1.2 内置函数的进一步理解

内置函数的用处远比一般的成员方法多得多。其左右在 Scripts/introduction 文件下的几个脚本中有所体现。下面归纳几个常用的内置函数的作用。

- **__getitem__**

  实现了该方法的类将自带一个迭代器，也即我们可以使用 for x in object 这类的迭代器操作，同时也可以使用 object[i] 这样类似 C++ 通过下标取值的操作。甚至于可以调用标准库的随机函数进行随机选取的操作。

- **__repr__**

  该方法的实现，能把一个对象用字符串的形式表达出来以便辨认。如果没有实现该方法，在控制台打印实例时将出现 <xxx object at 0x.....> 的形式。对应的还有一个 __str__ 内置方法，该方法实在 str() 函数被调用时使用，或是使用 print() 打印对象时使用，如果 __str__ 没有被实现，在相关操作中将会调用 __repr__ 替代。

这里只列举了两个例子来说明内置函数的强大之处，更多例子在原书中有解释，但原书也并没有对全部内置函数解释，各个内置函数的具体作用还需要在实践使用中理解。

```python
# 例1-2: 几个内置函数的理解
from math import hypot
import math


class Vector:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def __repr__(self):       # repr() 内置函数，以字符串形式表达，他和 str() 不同
        return 'Vector(%r,%r)' % (self.x, self.y)

    def __abs__(self):        # abs() 内置函数
        return math.sqrt(self.x**2 + self.y**2)

    def __bool__(self):       # 判断是否为空时调用
        return bool(abs(self))

    def __add__(self, other): # 加法运算
        x = self.x + other.x
        y = self.y + other.y
        return Vector(x, y)

    def __mul__(self, scalar): # 数乘运算
        return Vector(self.x * scalar, self.y * scalar)
```

II 数据结构

2 序列组成的数组

2.1 内置序列类型概览

首先复习几个基础概念，可将 Python 基础数据类型分为以下两序列类：

- **容器序列**

  list、tuple、collections.deque

- **扁平序列**

  str、bytes、bytearray、memoryview、array.array

容器序列存放的时它们所包括的任意类型的对象的引用，扁平序列李存放的是值而不是引用。换句话说，扁平序列其实是一段连续的内存空间，扁平序列其实更加紧凑，但是它里面只能存放诸如字符，字节和数值这种基础类型。

同时，序列类型还能按照能否被修改来分类。

- **可变序列**

  list、bytearray、array.array、collections.deque、memoryview

- **不可变序列**

  tuple、str、bytes

下图显示了可变序列 (MutableSequence) 与不可变序列 (Sequence) 的继承关系。虽然内置的序列类型不是直接继承自这两个抽象类，但了解这些基类可以帮助我们理解那些完整的序列类型包含了哪些功能。

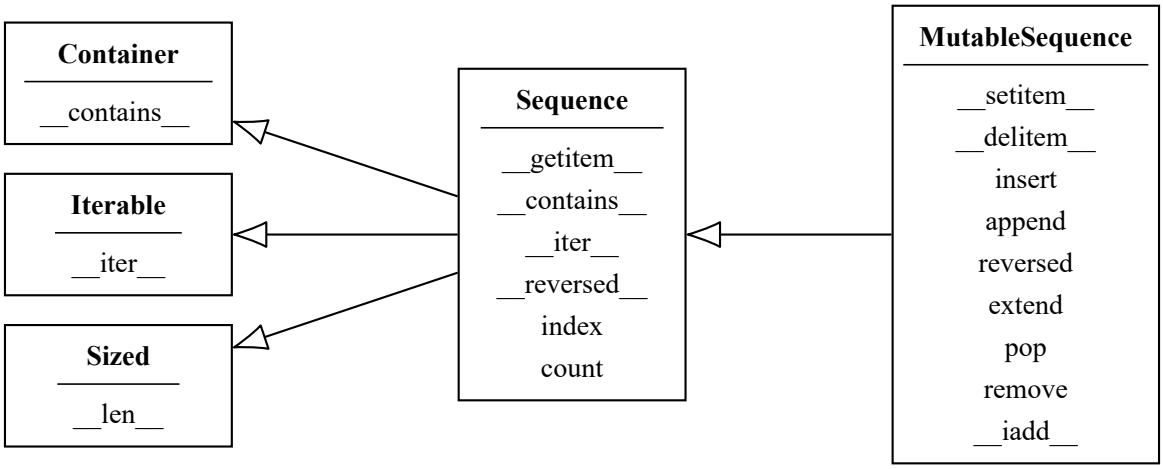

图 2.1 序列继承关系

2.2 列表推导和生成器表达式

列表推导是构建列表 (list) 的快捷方式，生成器表达式可以用来创建其他任何类型的序列。

2.2.1 列表推导和可读性

下面两种写法分别用常规方法和列表推导创建了新的列表，可以看出使用列表推导更加方便。

```
# 例2-1: 使用 for 循环建立新列表
symbols = '$€£$€£'
codes = []
for symbol in symbols:
    codes.append(ord(symbol))

print(codes)
```

```
# 例2-2: 使用列表推导创建新列表
symbols = '$€£$€£'
codes = [ord(symbol) for symbol in symbols]

print(codes)
```

列表推导有时也可能被滥用，使得代码可读性极差。通常的原则是：只用列表推导来创建新的列表，并且保证尽量精简。如果列表推导代码超过了两行，那可以考虑用外部 for 循环来替代了。

在 Python 2.x 版本中，列表推导可能会造成变量泄泄漏的问题，即在列表推到中使用到的变量会影响到列表推导外的变量。不过在 python3.x 中已经修复了这一问题，列表推导中的变量将被视为局部变量。

2.2.2 列表推导进阶用法

列表推导可以帮助我们把一个序列或者其他可迭代类型的元素进行过滤或加工，再生成一个新的列表。使用 Python 内置的 filter，map 结合 lambda 表达式也可以达到类似的效果，但语法比较复杂，可读性很差。

```
# 例2-3: 比较 filter/map 与 列表推导
symbols = '$€£$€£'

ascii1 = [ord(s) for s in symbols if ord(s) > 127]
ascii2 = list(filter(lambda c: c > 127, map(ord, symbols)))
```

并且在上述代码中，使用列表推导的速度不一定比 map/filter 组合慢。

同样的，列表推导中也可以嵌套循环，其作用和在外部使用 for 循环类似，下面是书上的例子。

```python
# 例2-4: 列表推导笛卡尔积
colors = ['black', 'white']
sizes = ['S', 'M', 'L']

tshirts1 = [(color, size) for color in colors for size in sizes] # 先颜色后尺码
tshirts2 = [(color, size) for size in sizes for color in colors] # 先尺码后颜色
```

2.2.3 生成器表达式

列表推导的作用只有一个：生成列表。如果需要生成其他类型的序列，需要用到生成器表达式。虽然列表推导也可以间接创建元组等其他序列，但在内部处理过程中，列表推导会创建一个完整的列表，再将其传递到某个构造函数中。生成器表达式则不同，生成器表达式遵循了迭代式协议，可以逐个产出元素，这显然对内存更加友好。

```python
# 例2-5: 用生成器表达式创建元组和数组
import array

symbols = '$€£$€£'
ascii = tuple(ord(symbol) for symbol in symbols)
sen = array.array('I', (ord(symbol) for symbol in symbols))
```

在创建 sen 的过程中，在第二个参数 (即生成器表达式) 外部加了一个括号，这是因为：如果生成器表达式时一个函数调用过程中的唯一参数，那么不需要格外括号，否则需要。

下面给出一个笛卡尔积的例子：

```python
# 例2-6: 生成器表达式计算笛卡尔积
colors = ['white', 'black']
sizes = ['M', 'L', 'S']

for tshirt in ('%s %s' % (c, s) for c in colors for s in sizes):
    print(tshirt)
```

在上述例子中，生成器表达式并不会在内存中留下一个有 6 个组合的列表，因为生成器表达式会在每次 for 循环运行时才生成一个组合。如果数据量更大，这样处理的优势将更加明显。

2.3 元组不仅是不可变列表

2.3.1 元组和记录

元组其实是对数据的记录：元组中的每个元素都存放了记录中一个字段的数据，外加这个字段的位置。正式位置信息给数据赋予了意义。

```python
# 例2-7: 将元组作为记录
lax_coordinate = (33.19, -118.40)
city, year, pop, chg, area = ('Tokyo', 2003, 32450, 0.66, 8014)
```

```
traveler_ids = [('USA', '31195855'), ('BRA', 'CE342567'), ('ESP', 'XDA205856')]

for passport in sorted(traveler_ids):
    print('%s,%s' % passport)    # % 运算符匹配到对应元组变量上
for country, _ in traveler_ids:  # 第二个元素没什么用，用 _ 占位
    print(country)
```

2.3.2 元组拆包

在例 2-7 第 3 行，我们指用一步就将 ('Tokyo', 2003, 32450, 0.66, 8014) 赋值给了 city, year, pop, chg, area 五个变量。同样在第 7 行，一个% 运算符就把 passport 元组里的元素对应到了 print 函数的格式化字符串中。这都是元组拆包。

元组拆包可用在任何可迭代对象中，唯一硬性要求是：被可迭代对象中的元素数量必须要跟接受这些元素的元组空挡数一致。除非采用 * 表示忽略多余元素[1]。下面是几种元组拆包的方法：

- **平行赋值**

  平行赋值也即例 2-7 第 3 行用到的方法，即把一个可迭代对象里的元素，一并赋值到由对应的变量组成的元组中：

```
lax_coordinate = (33.95,-118.40)
latitude,longitude = lax_coordinate
```

- **交换变量**

  利用这种思想可以在不添加中间变量交换变量值

```
a,b = b,a
```

- **_ 占位符**

```
        import os
        _,filename = os.path.spilt('/home/pionpill/.ssh/idrsa.pub')
```

  python 许多函数返回值为元组，那么我们就可以使用平行赋值的方法获取其中的元素，对不感兴趣的元素可以使用 _ 占位符。

- *** 拆包**

  * 的一个用法是把可迭代对象拆开作为函数参数

```
t = (20,8)
div(*t)    # 等同于 div(20,8)
```

  此外，*args 也可用于获得不确定数量的参数，以列表形式返回。

```
a,b,*rest = range(5)  # a,b,*rest 分别为 (0,1,[2,3,4])
a,b,*rest = range(3)  # a,b,*rest 分别为 (0,1,[2])
a,b,*rest = range(2)  # a,b,*rest 分别为 (0,1,[])
```

[1]后文将详细讲解

在平行赋值中，* 前缀只能出现在一个变量名前，但可以出现在任意位置

```
a,*body,c,d = range(5) # (0,[1,2],3,4)
```

2.3.3 嵌套元组拆包

接受表达式的元组可以是嵌套的，只要这个接受元组的嵌套结构符合表达式本身的嵌套结构。

```
# 例2-8: 嵌套元组获取经度
metro_areas = [
    ('Tokyo', 'JP', 36.9, (35.6, 139.69)),
    ('Delhi NCR', 'IN', 21.9, (28.6, 77.2)),
    ('Mexico City', 'MX', 20.1, (19.4, -99.1)),
    ('New York-Newark', 'US', 20.1, (40.8, -74.0)),
]

fmt = '{:15}|{:9.1f}|{:9.1f}'
for name, cc, pop, (latitude, longitude) in metro_areas:
     if longitude <= 0:
         print(fmt.format(name,latitude,longitude))
```

上例第 8 行，将元组中最后一个元素拆包到由变量构成的元组里，这样就获得了坐标。

2.3.4 具名元组

collections.namedtuple 是一个工厂函数，可以用来构建一个带字段名的元组和一个有名字的类。且用 namedtuple 构建的类的实例消耗的内存和元组是一样的。

```
# 例2-9: 具名元组
from collections import namedtuple

City = namedtuple('City', 'name country population coordinates')
tokyo = City('Tokyo', 'JP', '36.9', (35.6, 139.6))
print(tokyo.name)    # 'JP'
print(tokyo[1])      # 'JP'
```

在上例中，第 4 行创建一个具名元组需要两个参数，一个是类名，另一个是各个字段的名字。后者可以是由数个字符串组成的可迭代对象，或者是由空格分开的字符名组成的字符串。

除了从最普通元组那里继承来的属性外，具名元组还有一些自己专有的属性。

```
# 例2-10 具名元组的属性和方法
from collections import namedtuple

City = namedtuple('City', 'name country population coordinates')
print(City._fields)   # ("name","country","population","coordinates")

LatLong = namedtuple('LatLong', 'lat long')
```

```
8  delhi_data = ('Delhi NCR', 'IN', '21.9', LatLong(28.6, 77.2))
9  delhi = City._make(delhi_data)
10 for key, value in delhi._asdict().items():
11     print(key+':', value)
```

下面结合上例对几个常用的属性解释：

- _fields 属性

  第 5 行，返回一个元组，包含类中的所有字段名

- _make() 方法

  第 9 行，接收一个可迭代对象生成这个类的一个实例，相当于 City(*delhi_data)。

- _asdict() 方法

  将具名元组以 collections.OrderedDict 形式返回，结合 item() 以列表形式返回。

2.3.5 作为不可变列表的元组

元组与列表非常相似，除了跟增减方法有关的方法外，元组支持列表的其他所有方法[2]。这其中有一个列外：元组没有 __reversed__ 方法，但这并不妨碍对元组使用 reversed(tuple) 方法，因为 __reversed__ 方法仅是实现 reversed() 的一种优化内置函数。

2.4 切片

切片的基础知识这里不再讲解，读者可以自行阅读书 2.4.1-2.4.2,2.4.4 的内容。

2.4.1 多维切片和省略

[] 除了一维的切片运算，还可以使用以逗号分开的多个索引或者是切片。例如 Numpy 中二维的 numpy.ndarray 就用到了 a[i,j] 抑或是 a[m:n,k:l] 的形式。

要正确处理这种 [] 运算符的话，对象的特殊方法 __getitem__ 和 __setitem__ 需要以元组的形式来接收 a[i,j] 中的索引。也就是说，如果要得到 a[i,j] 的值，Python 会调用 a.__getitem__((i,j))。

省略 (ellipsis) 的写法是...(三个英文句号)。省略在 Python 解析器中是一个符号，实际上它是 Ellipsis 对象的别名，Ellipsis 对象是 ellipsis 类的单一实例。它可以用作切片规范的一部分或函数清单中，例如在 Numpy 中对某个思维数组 x 采用 x[1,...] 操作等同于 x[1,:,:,:]。

2.5 对序列使用 + 和 *

通常 + 号两侧的序列由相同类型的数据所构成，在拼接过程中，两个操作的序列都不会被修改，Python 会新建一个包含同样类型数据的序列来作为拼接的结果。

+ 和 * 都遵守这个规则，不修改原有的操作对象，而是构建一个全新的序列。

[2] 具体的方法见书

如果在 a*n 这个语句中，序列 a 里的元素是对其他可变对象的引用的话，就会出问题，如下面例子。

```
# 例2-12: 利用*创建二维列表
board1 = [['_'] * 3 for i in range(3)] # 创建一个 3x3 列表
board2 = [['_'] * 3] * 3
board3 = ['_'] * 3 * 3

board1[0][0] = [1]       # [1,'_','_'],['_','_','_'],['_','_','_']
board2[0][0] = 1         # [1,'_','_'],[1,'_','_'],[1,'_','_']
board3[0] = 1            # [1,'_','_','_','_','_','_','_','_']
```

在上面这个例子中第 7 行我们会发现，对 board[0][0] 的修改导致每列第一个值都进行了修改。这是因为当 a*n 语句中 a 中元素为引用时，列表新增元素也将指向这个引用[3]。

上述例子的写法等同于下例：

```
# 例2-13: 等效外部 for 循环
board1 = []
for i in range(3):
    row1 = ['_'] * 3
    board1.append(row1)

row2 = ['_'] * 3
board2 = []
for i in range(3):
    board2.append(row2)
```

2.6 序列的增量赋值

增量赋值运算符 += 和 *= 的表现取决于它们的第一个操作对象。下面只讨论 +=。

+= 背后的方法是 __iadd__。表示就地加法，但是如果没有实现该方法，将会退一步调用 __add__，得到一个新的对象，再赋值给原对象。也就是说在表达式中，变量名会不会关联到新的对象，完全取决于这个类型有没有实现 __idd__ 方法。

总的来说，可变序列都实现了 __iadd__ 方法，而不可变序列根本就不支持这个操作。

```
l = (1,2,3)
id1 = id(l)
l *= 2
id2 = id(l)
id1 == id2        # False
l = [1,2,3]
id1 = id(l)
l *= 2
id2 = id(l)
id1 == id2        # True
```

[3]内部原理将在后续章节讲解

一个关于 += 的谜题

运行下列代码：

```
# 例2-14：关于 += 的谜
a = (1, 2, [3, 4])
a[2] += [5, 6]        # (1,2,[3,4,5,6])
```

上述代码运行时会报错，但同时元组 a 将会改变[4]就能避免异常。如果我们利用 dis 查看汇编过程会发现，在上述代码运行过程中，先对 a[2] 指向的列表进行了修改，再将该列表传给了 a[2]。在传递过程中，由于元组不可被修改，故报错，但在实际打印 a 的值时，a[2] 所指向的列表却发生了改变。总结如下：

- 不要把可变对象放在元组里。
- 增量赋值不是一个原子操作。就像这里即使抛出异常，却完成了操作。
- 这种情况很罕见，原作者 15 年 Python 生涯都没遇到过。

2.7 list.sort 方法和内置函数 sorted

list.sort 会就地排序，不会把原列表复制一份。这也是这个方法返回值是 None 的原因[5]。与之相对的，内置函数 sorted 会新建一个列表作为返回值。这个方法可以接收任何形式的可迭代对象作为参数[6]。

这两个方法/函数都有两个可选的关键字参数：

- **reverse**

  如果被设定为 True，将降序排列，默认为 False。

- **key**

  一个只有一个参数的函数，这个函数会被用在序列里的每一个元素上，所产生结果是排序算法依赖的对比关键字。比如说用 key=str.lower 来实现忽略大小写排序。默认为恒等函数，即元素本身的值。

2.8 用 bisect 来管理已排序的序列

bisect 模块包括两个主要函数，bisect 和 insort，两个函数都利用二分查找算法来在有序序列中查找或插入元素。

[4] 如果调用 a[2].extend([50,60])

[5] Python 惯例：一个函数或方法对对象就行就地改动，那它的返回值应该是 None

[6] 甚至有些不可迭代对象，后文将提到

2.8.1 用 bisect 来搜索

`bisect(haystack,needle)` 在 `haystack`(干草垛) 里搜索 `needle`(针) 的位置，该位置曼珠的条件为：将 `needle` 插入这个位置后，`haystack` 仍然保持升序[7]。

如果需要进行排序操作，可以使用 `bisect(haystack,needle)` 来查找位置 `index`，再用 `haystack.insert(index,needle)` 插入新值。当然也可以使用 `insort` 一步到位。

```
# 例2-17: 在有序序列中用 bisect 查找某个元素位置
import bisect
import sys

HAYSTACK = [1, 4, 5, 6, 8, 12, 15, 21, 23, 23, 26, 29, 30]
NEEDLES = [0, 1, 2, 5, 8, 10, 22, 23, 29, 30, 31]

ROW_FMT = '{0:3d} @ {1:3d} {2}{0}'


def demo(bisect_fn):
    for needle in reversed(NEEDLES):
        position = bisect_fn(HAYSTACK, needle)
        offset = position * ' |'
        print(ROW_FMT.format(needle, position, offset))


if __name__ == '__main__':
    if sys.argv[-1] == 'left':
        bisect_fn = bisect.bisect_left
    else:
        bisect_fn = bisect.bisect

    print('DEMO', bisect_fn.__name__)
    print('haystack ->', ''.join('%3d' % n for n in HAYSTACK))
    demo(bisect_fn)
```

bisect 还有两个可选参数，`lo,hi` 用来搜小搜寻范围，lo 默认值是 0，hi 默认值是序列长度。

其次，`bisect` 函数其实是 `bisect_right` 函数的别名，对应的还有一个 `bisect_left` 函数，在 `bisect_left` 函数中，插入的值若与原序列中某个值相同时，会被放在左边。

下面给出了一个 `bisect` 函数的实用例子：

```
# 例2-18: bisect 函数的一个应用
import bisect


def grade(score, breakpoints=[60, 70, 80, 90], grade='FDCBA'):
    i = bisect.bisect(breakpoints, score)
    return grade[i]

```

[7]前提是 haystack 是有序序列。

```
9
10 for score in [33, 99, 77, 70, 89, 90, 100]:
11     print(grade(score))
```

2.8.2 用 bisect.insort 插入新元素

`insort(seq,item)` 把变量 `item` 插入到序列 `seq` 中，并且能保持 `seq` 升序顺序。

```
1  # 例2-19: insort 排序
2  import bisect
3  import random
4
5  SIZE = 7
6
7  random.seed(1727)
8
9  my_list = []
10 for i in range(SIZE):
11     new_item = random.randrange(SIZE * 2)
12     bisect.insort(my_list, new_item)
13     print('%2d ->' % new_item, my_list)
```

与 `bisect` 类似的，`insort` 也有 `lo` 和 `hi` 两个可选参数来控制查找范围，`insort` 也有个变体叫 `insort_left`。

目前提到的内容不仅适用于列表或元组，还可用于几乎所有的序列类型。

2.9 当列表不是首选

虽然列表灵活又简单，但只需要处理数字列表的话，会有更好的选择。比如存放 100 万的浮点数时，使用数组 (array) 效率要高得多，因为数组存放的并不是浮点数对象，而是数字的机器翻译，也就是字节表述[8]。

2.9.1 数组

如果只需要包含数字的列表，那么 `array.array` 比 list 更高效。数组支持所有和可变序列有关的操作。另外，数组还提供从文件读取和存入文件的更快的方法。

Python 数组和 C 语言一样，创建数组需要一个类型码，这个类型码规定了在底层的 C 语言应该处理什么样的数据类型。这样可以节省很多空间。

```
1  # 例2-20: 数组使用
2  from array import array
3  from random import random
4
5  floats = array('d', (random() for i in range(10**7)))
```

[8]这和 C 语言处理数组一样

```
print(floats[-1])
```

在书上示例中，还用到了处理数据文件的方法，使用二进制方式存储文件即节省了空间，Python 在处理二进制文件时读取速度相较文本文件也快许多。

2.9.2 内存视图

memoryview 是一个内置类，它能让用户在不复制内容的情况下操作同一个数组的不同切片。

memoryview.cast 的概念跟数组模块类似，能用不同的方式读写同一块内存数据，而且内容不会随意移动，这听上去和 C 语言的强制类型转化概念相似。memoryview.cast 会把同一块内存里的内容打包成一个全新的 memoryview 对象给你。

```
# 例2-21: 通过改变数组中一个字节来更新数组里某个元素值
import array

numbers = array.array('h', [-2, -1, 0, 1, 2])
print(numbers)           # array('h', [-2, -1, 0, 1, 2]): 原始 numbers

memv = memoryview(numbers)
print(len(memv))         # 内存视图的基础操作
print(memv[0])

memv_oct = memv.cast('B')
# [254, 255, 255, 255, 0, 0, 1, 0, 2, 0]: 将机器码转换为整型后以列表形式输出
print(memv_oct.tolist())
memv_oct[5] = 4
# array('h', [-2, -1, 1024, 1, 2]): 内存处机器码改变，numbers 也改变
print(numbers)
```

如果要进行高级的数据处理，那可以继续学习 NumPy 和 SciPy 两个外部库。

2.9.3 双向列表和其他形式的队列

利用 .append 和 .pop 方法，列表也能实现栈或者队列的操作。但是删除列表的第一个元素抑或是在列表后面添加元素之类的操作时很耗时的，因为这些操作会牵扯到移动列表中的所有操作。

collections.deque 类 (双向队列) 是一个线程安全，可以快速从两端添加或删除元素的数据类型。在新建双向队列时，可以指定队列大小，如果队列满员了，还可以从方向端删除过期的元素，然后再尾端添加新的元素。

```
# 例2-23: 双向队列
from collections import deque

dq = deque(range(10), maxlen=10)
dq.rotate(3)                    # 队列右移
dq.rotate(-4)                   # 队列左移
```

```
dq.appendleft(-1)                # 队列头部插入-1，尾部最后一个元素删除
dq.extend([11, 22, 33])          # 尾部添加3个元素，头部删除三个元素
dq.extendleft([10, 20, 30, 40]) # 头部逐个添加4个元素
```

双向队列实现了大部分列表所拥有的方法，也有一些格外的符合自身设计的方法，比如`popleft`，`rotate`。但是为了实现这些操作，双向队列也付出了一定代价，从队列中间删除元素的操作会慢一些，因为它只在对头部的操作进行了操作。

除了 deque 之外，Python 标准库还提供了其他几种对队列的实现，这里不再一一解释。如`queue`,`multiprocessing`,`asyncio`,`heapq`。

3 字典和集合

dict 类型是 Python 语言的基石，即使我们没有直接使用到字典。和它相关的内置函数在 __builtins__，__dict__ 模块中。Python 对实现字典进行了高度优化，散列表[9]是字典类型性能出众的根本原因。集合 (set) 的实现也依赖于散列表。

3.1 泛映射类型

`collections.abc` 模块中有 `Mapping` 和 `MutableMapping` 两个抽象基类[10]，它们的作用是和其他类似的类型定义形式接口。

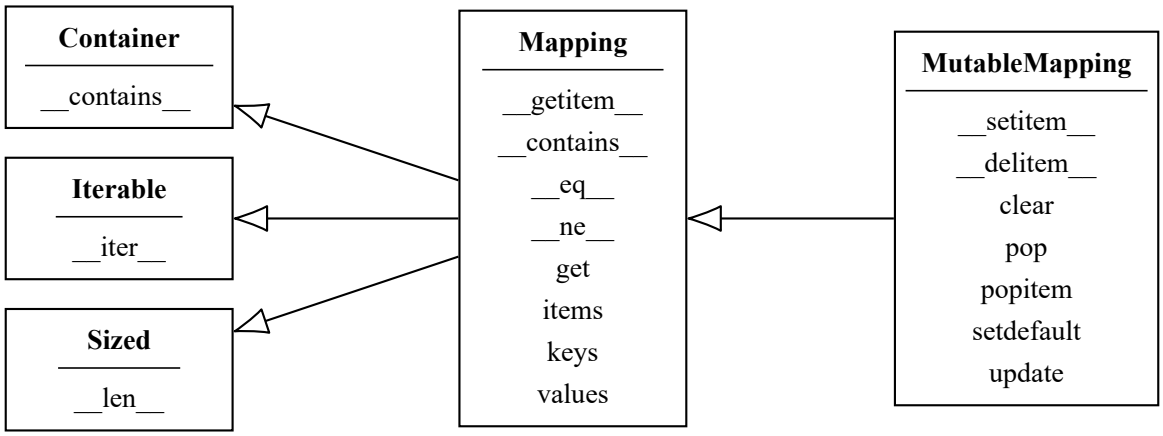

图 3.1 MutableMapping 继承关系

非抽象映射类型一般不会直接继承这些继承这些抽象基类,而是会继承 `dict` 或是 `collections.Use` 进行扩展。可以使用 isinstance 一起被用来判定某个数据是不是广义上的映射类型。

```
my_dict = {}
isinstance(my_dict,abc.Mapping) # True
```

Python 标准库里的所有映射类型都是利用 dict 来实现的，因此它们有个共同的限制，即只有可散列的数据类型才能用作这些映射里的键 (只有键有限制，值没有)。

可散列对象需要满足以下三个条件：

1. 在生命周期中，散列值不变。
2. 需要实现 __hash()__ 方法。
3. 需要实现 __eq()__() 方法，通过 __id()__ 查散列值用于和其他键作比较。

在 Python 中，有三类数据类型是可散列的：

1. 原子不可变数据类型 (str,bytes 和数值类型)。
2. `frozenset` 是可散列的，因为它只能容纳可散列对象。
3. 元组，当元组内部包含的所有元素都是可散列元素。

[9]别名：哈希表

[10]在 Python 2.6-3.2 中，这两个类属于 `collections` 模块

字典的构造方法有许多，例如如下例子：

```
a = dict(one=1,two=2,three=3)
b = {'one':1,'two':2,'three':3}
c = dict(zip(['one','two','three'],[1,2,3]))
d = dict([('two':2),('one':1),*('three':3)])
e = dict({'three':3,'one':1,'two':2})
a == b == c == d == e     # True
```

3.2 字典推导

字典推导的概念从 Python 2.7 起移植自列表推导。字典推导可以从任何以键值对作为元素的可迭代对象中构建出字典。

```
# 例3-1: 字典推导
DIAL_CODES = [(86,'China'),(91,'India'),(1,'United States'),(62,'Indonesia')]
country_code = {country:code for code,country in DIAL_CODES}
print(country_code)
```

字典推导的用法和列表推导几乎一致，这里不再赘述。

3.3 常见的映射方法

用 setdefault 处理找不到的键

当字典 `d[k]` 不能找到正确的键的时候，可以通过 `d.get(k,default)` 来给找不到的键一个默认的返回值。但是要更新某个键对应的值的时候，不管使用 `__getitem__` 还是 `get` 都不自然，且效率低。这时可以用 `dict.setfault` 代替。

```
    # 写法一: setdefault
    my_dict.setdefault(key,[]).append(new_value)
    # 写法二: 常规写法
    if key not in my_dict:
    my_dict[key] = []
    my_dict[key].append(new_value)
```

以上两种写法效果相同，第一种只需一次查询可以完成整个操作，二第二种需要至少两次查询，如果键不存在，则需要三次。

3.4 映射的弹性键查询

有时候为了方便起见，就算某个键在映射里不存在，我们也希望通过这个键能读取到一个默认值。有两种方法达到这个目的，一是使用 `defaultdict` 这个类型。二是自定义一个 `dict` 的子类，实现 `__missing__` 方法。

3.4.1 defaultdict：处理找不到的键的一个选择

在实例化一个 collections.defaultdict 的时候，需要为构造方法提供一个可调用对象，这个可调用对象会在 __getitem__ 碰不到键的时候被调用，返回默认值。

例如我们新建这样一个字典：dd = defaultdict(list)，如果 'new-key' 在 dd 中还不存在，表达式 dd[`new-key'] 会按以下步骤来行事：

1. 调用 list() 来建立一个新列表。
2. 把这个新列表作为值，`new-key' 作为键，放到 dd 中。
3. 返回这个列表的引用。

这个用来生成默认值的可调用对象存放在名为 default_factory 的实例属性里。如果在创建 defaultdict 时没有指定 default_factory，查询不存在的键会触发 KeyError。用法见下例[11]。

```
# 例3-5: defaultdict 的使用
import collections

bag = ['apple','banana','peach','watermelon','apple','banana','apple']
count = collections.defaultdict(int) # int() 生成 0
for fruit in bag:
    count[fruit] += 1
# defaultdict(<class 'int'>, {'apple': 3, 'banana': 2, 'peach': 1, 'watermelon': 1})
```

default_factory 只会在 __getitem__ 里被调用，在其他方法中不会发挥作用。

3.4.2 特殊方法 __missing__

所有映射类型在处理找不到的键的时候，都会牵扯到 __missing__ 方法。如果一个类继承了 dict，然后提供了 __missing__ 方法，在 __geiitem__ 碰到找不到键的时候，Python 会自动调用它，而不是抛出 KeyError 异常。

__missing__ 方法只会被 __getitem__ 调用，这也是上一节 default_factory 只会被 __getitem__ 调用的原因。

有时候，在查询的时候，我们希望将键通通转换为 str。

```
# 例3-7: 将非字符串键转换为字符串
class StrKeyDict0(dict):         # 继承自 dict

    def __missing__(self,key):
        if isinstance(key,str):  # 如果找不到的键是字符串，抛出异常
            raise KeyError(key)
        return self[str(key)]    # 如果找不到的键不是字符串，转换成字符串再查找

    def get(self,key,default=None):
        ''' get 方法把查找工作用 self[key] 形式委托给 __getitem__,这样在查找失败后还能通过
            __missing__ 再给键一个机会 '''
```

[11]例 3-5 为本人自己编写，参考文章：CSDN:yealxxy

```
11          try:
12              return self[key]
13          except KeyError:
14              return default       # 如果抛出 KeyError 说明 __missing__ 也失败了，返回 default
15
16      def __contains__(self, key):
17          return key in self.keys() or str(key) in self.keys()
18          # contains 并不会调用 __getitem__，也就不会调用 __missing__ 因此要使用 or
```

在上述代码中，如果没有第 5 行判断，调用 `__gititem__` 则会陷入无限循环。在第 17 行中使用了 `self.keys()`[12] 而不是 `for key in dict`，因为这样会导致递归调用，可能陷入无限循环。

3.5 字典的变种

这一节主要讲 `collections` 模板中，除了 `defaultdict` 之外的映射类型。

- `collections.OrderedDict`
  添加键的时候保持顺序。`popitem` 方法默认删除并返回字典的最后一个元素。
- `collections.ChainMap`
  该类型可以容纳数个不同的映射对象，在进行键查找操作的时候，对象会被当作一个整体逐个查找。
- `collections.Counter`
  该映射类型会给键准备一个整数计数器。每次更新一个键的时候都会增加这个计数器。

```
1       ct = collections.Counter('abcdeabcdabc') #
            Counter({'a':3,'b':3,'c':3,'d':2,'e':1})
```

- `collections.UserDict`
  把标准 `dict` 用纯 Python 又写了一遍。

3.6 子类化 UserDict

在创造自定义映射类型来说，用 `UserDict` 比普通的 `dict` 作为基类更方便。主要原因是，后者在某些方法的实现上会走一些捷径，导致我们不得不在它的子类中重写这些方法[13]，但是 `UserDict` 并没有这些问题。

`UserDict` 并不是 `dict` 的子类，但是 `UserDict` 有一个 `data` 属性，是 `dict` 实例，这个属性实际上是 `UserDict` 最终存储数据的地方。这样做的好处是，比起例 3-7，`UserDict` 的子类能在实现 `__setitem__` 的时候避免不必要的递归，也可以让 `__contains__` 里的代码更简洁。

```
1   # 例3-8: 继承自 UserDict 的映射类
```

[12]这种操作在 Python3 中很快，不同担心速度问题
[13]关于内置类继承有什么不好，在后面章节会说明

```
import collections

class StrKeyDict(collections.UserDict):
    def __missing__(self, key):
        if isinstance(key,str):
            raise KeyError(key)
        return self(str(key))

    def __contains__(self, key):
        return str(key) in self.data # data 中所有键均为 str

    def __setitem__(self, key, item):
        self.data[str(key)] = item   # 具体的实现委托给了 data 属性
```

`UserDict` 继承自 `MutableMapping`，里面有两个常用的方法值得关注：

- `MutableMapping.update`

  该方法不但可以为我们所直接使用，还用在 `__init__` 中，让构造方法可以利用传入各种参数来新建实例。这个方法的背后是 `self[key] = value` 来添加新值，所以它其实是在使用 `__setitem__` 方法。

- `Mapping.get`[14]

  在示例 3-7 中，我们改写了 `get` 方法，然而在例 3-8 中却没有必要，因为 `Mapping.get` 方法的实现和例 3-7 中的实现一致。

3.7 不可变映射类型

标准库中的所有映射类型都是可变的，从 Python 3.3 开始，`types` 模块引入了一个封装类 `MappingProxyType`。如果给这个类一个映射，它会返回一个只读的映射视图。虽然是只读的，但它是动态的。这意味着如果对原映射做了改动，通过这个视图可以观察到，但是无法通过这个视图对原映射做出修改。

```
# 例3-9: 字典只读实例 mappingproxy
from types import MappingProxyType

d = {1:'A'}
d_proxy = MappingProxyType(d) # mappingproxy instance {1: 'A'}
# d_proxy[2] = 'x'           # Error: 不能通过 d_proxy 修改
d[2] = 'B'                   # d_proxy 是动态的，可以通过 d 修改 d_proxy 视图
print(d_proxy)               # mappingproxy instance {1: 'A', 2: 'B'}
```

[14]Mapping 是 MutableMapping 的父类

3.8 集合论

Python 中的“集”[15]指 `set` 和它的不可变姐妹类型 `frozenset`, 它们在 Python 2.6 才被列入内置类型，属于非常年轻的概念。

集合的本质是许多唯一对象的聚集。集合中的元素必须是可散列的，`set` 类型本身是不可散列的，但是 `forzenset` 可以。

集合实现了很多基础的中缀运算，利用这运算可以简化代码，同时还能减少程序运行时间，给定两个集合 `a` 和 `b`。

- `a|b`: 合集
- `a&b`: 交集
- `a-b`: 差集

下面看一个例子，假定有两个地址集合，一个较小的 `needles`，一个较大的 `haystack`：

```
# 写法一
found = len(needles & haystack)
# 写法二
found = 0
for n in needles:
    if n in haystack:
        found += 1
# 写法三
found = len(set(needles).intersection(haystack))
```

其中写法一比二速度要快一些，但写法二可以用在任何可迭代对象上。写法三适用于所有可迭代对象，但会牵扯到把对象转换为集合的成本，不过 `needles` 或是 `haystack` 若任何一个已经是集合，效率仍然会更高。

3.8.1 集合字面量

除了空集之外，集合的字面量——{1},{1,2} 等，看起来和它的数学形式一模一样。如果是空集，必须写成 `set()` 形式。在创建空集过程中，也必须适用 `set()`，使用 {} 则会创建空字典。空集的返回形式也是 `set()`。

利用 `1,2,3` 这样的字面量句法比构造方法 `set([1,2,3])` 更快且更易读。后者在构建过程中，必须先从 `set` 这个名字来查询构造方法，然后新建一个列表，最后再把列表传入构造方法。如果是字面量，则会利用一个专门叫做 `BUILD_SET` 的字节码创建集合。可惜的是 frozenset 并没有字面量句法。

3.8.2 集合推导

集合推导和字典推导一样，源于列表推导，这里不再过多介绍

[15]原书中“集”指 set 和 frozenset，“集合”指 set

3.8.3 集合的操作

按照惯例，查看一下集合的继承关系

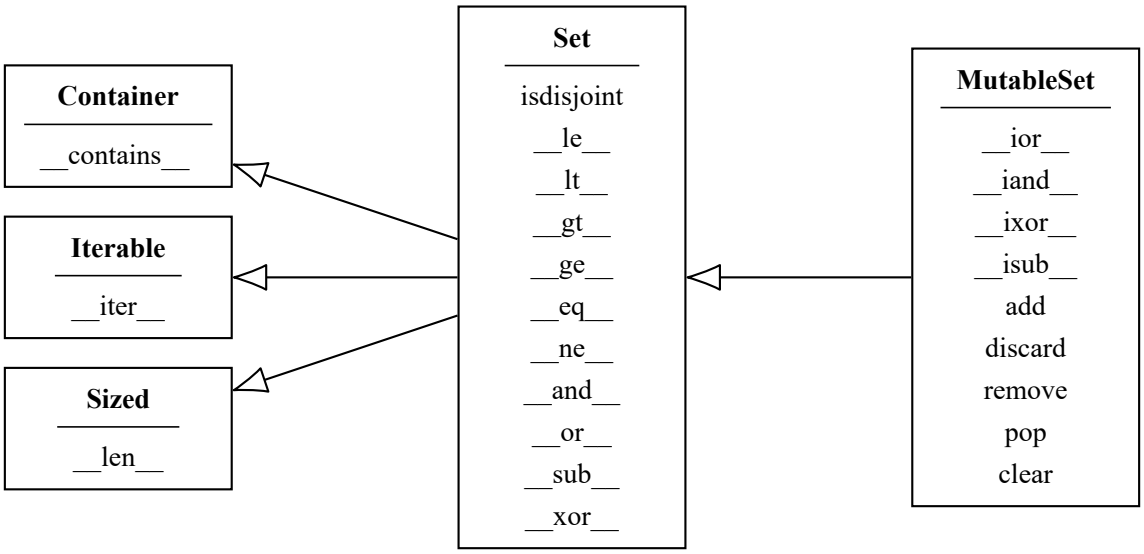

图 3.2 MutableSet 继承关系

集合有关的数学操作这里不再一一列举。

3.9 dict 和 set 的背后

3.9.1 一个关于效率的实验

从经验中可以得出结论，字典和集合的速度非常快。在作者的实验[16]中，作者从 1000 个元素的字典里搜索 1000 个浮点数用了 0.202 毫秒，在 10000000 个元素的字典里搜索 1000 个浮点数用了 0.337 毫秒。集合的速度也相差无几。

然而用到列表运算时，最后却用到了 97 秒，无疑列表的速度最为糟糕。这是因为列表背后没有散列表来支持 in 运算符。每次搜索都需要扫描一次完整的列表，这使得时间线性增加了。

3.9.2 字典中的散列表

这节简单介绍了散列表的原理。

散列表其实是一个稀疏矩阵，在 `dict` 的散列表中，每个键值对都占用一个表元，每个表元都有两个部分，一个是对键的引用，另一个是对值的引用。因为所有表元大小一致，所以可以通过偏移量来读取某个表元。

Python 会设法保证大概还有三分之一的表元是空的，所以在快要达到这个阈值时，原有的散列表会被复制到一个更大的空间里面。

[16]原书 P74-75

散列值和相等性

如果两个对象在比较的时候是相等的，那它们的散列值必须相等，否则散列表就不能正常运行了[17]。

为了让散列值能够胜任散列表索引这一角色，它们必须在索引空间中尽量分散开来。这意味着在理想状态下，越是相似但不相等的对象，散列值差别越大。

散列表算法

为了获取 `my_dict[search_key]` 背后的值，Python 首先会调用 `hash(search_key)` 计算 `search_key` 的散列值，把这个值最低的几位数字仿作偏移量，在散列表里查找表元。若找到的表元为空，则抛出 `KeyError` 异常。若不为空，则表元里一定会有一对 `found_key:found_value`。这时候会检验 `search_key == found_key`，若相等，则返回 `found_value`。

如果 `search_key` 与 `found_key` 不匹配，称为散列冲突。其原因是：散列表所做的其实是把随机的元素映射到只有几位的数字上，而散列表本身的索引又只依赖于这个数字的一部分。为了解决这个冲突，算法会在散列值中另外再取几位，然后用特殊的方法处理一下，把新得到的数字再当作索引来寻找表元。若这次得到的表元是空的，同样抛出 `KeyError`；若非空，或者键匹配，则返回这个值；若又发现了散列表冲突，则重复。

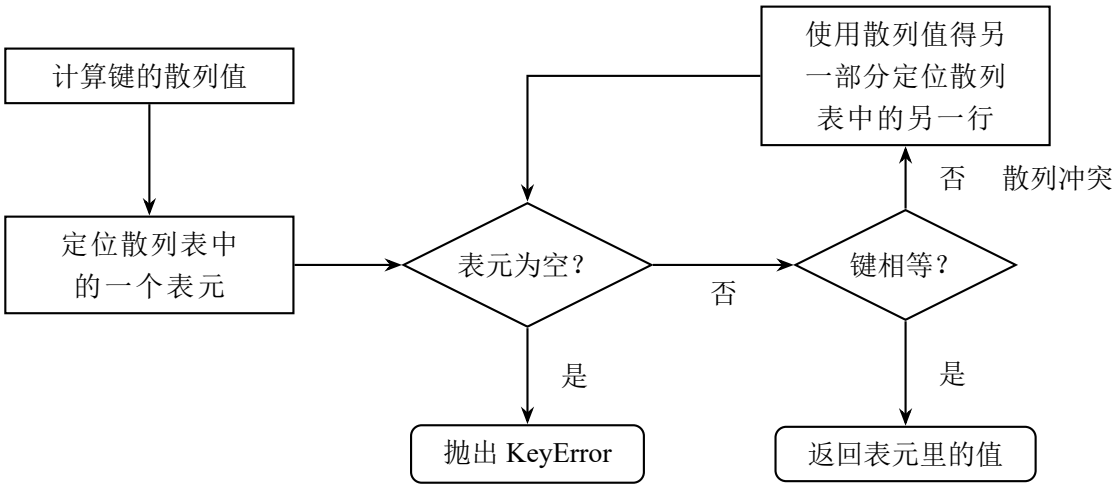

图 3.3　散列表原理

在插入新值时，Python 可能会按照散列表拥挤程度来决定是否要重新分配内存来扩容。这样做的目的是减少发生散列冲突的概率。正常情况下，即使是百万个元素，散列冲突的次数也不会超过 5 次。

3.9.3 `dict` 的实现及其导致的结果

键必须是可散列的

一个可散列的对象必须满足以下要求：

1. 支持 hash() 函数，并通过 `__hash__` 方法所得到的散列值是不变的。

[17]散列值相同，内部结构不一定相同

2. 支持通过 `__eq__` 方法来检验相等性。
3. 若 `a == b` 为真，则 `hash(a) == hash(b)` 也为真。

所有由用户自定义的对象默认都是可散列的，因为它们的散列值由 `id()` 获取，而且它们都是不相等的。

如果用户自定义了一个类的 `__eq__` 方法，并且希望它是可散列的，那么它一定要有个恰当的 `__hash__` 方法，保证 `a == b` 为真，则 `hash(a) == hash(b)` 也为真。否则就会破坏恒定的散列表算法，导致由这些对象所组成的字典和集合完全失去可靠性，这个后果非常可怕。另一方面，如果一个含有自定义的 `__eq__` 依赖的类处于可变的状态，那就不要在这个类中实现 `__hash__` 方法，因为它的实例时不可散列的。

字典在内存上开销巨大

由于字典使用了散列表，散列表又必须是稀疏的，这导致空间效率低下。因此大规模数据还是放在元组中较好，一方面这会节省空间，另一方面元组无需把记录中字段的名字在每个元素里都遍历一遍。

在用户自定义类型中，`__slots__` 属性可以改变实例属性的存储方式，由 `dict` 编程 `tuple`，后面章节会细讲。

空间优化在开发过程中资源足够的情况下并不那么重要，因为优化的对立面就是可维护性。

键的次序取决于添加顺序

在 `dict` 里添加新键而又发生散列冲突的时候，新键可能会被安排存放在另一个位置。于是会发生这样一种情况，`dict([(key1,value1),(key2,value2)])` 和 `dict([(key2,value2),(key1,value1)])` 得到的是两个字典。在进行比较的时候，它们是相等的；但如果 `key1` 和 `key2` 在添加到字典里的过程中又冲突发生的话，这两个键出现在字典里的顺序是不一样的。

往字典里添加新键可能会改变已有键的顺序

往字典里添加新的键，Python 解释器有可能会做出为字典扩容的决定，这个过程可能会发生散列冲突，并导致新散列表中键的次序发生变化。由此可知，不要对字典同时进行迭代和修改。如果必须这样做，最好分成两步：首先对字典迭代，得出想要修改的内容，并放在一个新的字典里；迭代结束后再对原有的字典进行更新。

3.9.4 set 的实现以及导致的结果

字典和散列表的几个特点对集合几乎都适用，为了避免集合有太多重复的内容，应遵循以下规则：

- 集合元素都是可散列的
- 集合很消耗内存
- 可以搞笑判断集合中是否存在某个元素
- 元素的次序取决于被添加到结合的次序
- 往集合里添加元素，可能会改变集合中元素的次序

4 文本和字节序列

4.1 字符问题

“字符串” 是个简单的概念：一串字符组成的序列，从 Python3 的 `str` 对象中获取的元素时 Unicode 字符，这相当于从 Python2 的 `unicode` 对象中获取的元素，而不是从 Python2 的 `str` 对象中获取的原始字符序列。

Unicode 标准把字符的标识和具体的字节表述进行了如下明确区分。

- 字符的标识，即码位，时 0-1114111 的数字 (十进制)，在 Unicode 标准中以 4-6 个十六进制数字表示，前缀 “U+”。
- 字符的具体表述取决于所用的编码，编码时在码位和字节序列之间转换时使用的算法。

把码位转换成字节序列的过程时编码，把字节序列转换成码位的过程时解码。

```
# 例4-1: 编码与解码
s = 'cafe'
b = s.encode('utf8')  # 使用 utf-8 将 'cafe' 编码成 bytes 对象
b.decode('utf8')      # 使用 utf-8 将 bytes 对象解码成 str 对象
```

4.2 字节概要

Python 内置了两种基本的二进制序列类型:Python3 引入的不可变的 `bytes` 类型和 Python2.6 添加的可变 `bytearray` 类型。`bytes` 和 `bytearray` 对象的各个元素时介于 0-255 之间的整数。

```
# 例4-2: bytes 和 bytearray 对象
cafe = bytes('café', encoding='utf_8') # b'caf\xc3\xa9'
print(cafe[0]) # 99
print(cafe[:1]) # b'c', bytes 对象的切片还是 bytes 对象

cafe_arr = bytearray(cafe)
print(cafe_arr) # bytearray(b'caf\xc3\xa9')
```

和许多序列类型一样，`seq[0]` 返回的是元素的类型，而 `seq[index:index+1]` 返回的则是 `seq` 类型，往往 `seq[0] == seq[:1]` 是成立的，但也仅对于 seq 这个类型成立。

二进制序列各个字节的值可能会使用以下三种不同的方式显示：

- 可打印的 ASCII 范围内的文字，使用 ASCII 本身，也即数字
- 制表符，换行符等，使用转义序列\t,\n。
- 其他字节类型，使用十六进制转义序列，例如 (\x00 是空字节)。

除了格式化方法和几个特殊处理 Unicode 数据的方法之外，`str` 类型的其他方法都支持二进制序列。`re` 模块的正则表达式也支持对二进制序列的处理。

二进制序列有个方法是 `str` 没有的，名为 `fromhex`，作用是解析十六进制数字对 (数字对

中间的空格可选)，构建新的二进制序列。

```
bytes.fromhex('31 4B CE A9') # b'1K\xce\xa9'
```

构建 bytes 或 bytearray 示例还可以调用各自的构造方法，传入下述参数：

- 一个 str 对象和一个 encoding 关键字参数。
- 一个可迭代对象，提供 0-255 之间的数值。
- 一个整数的，使用空字节创建对应长度的二进制序列 (已过失)。
- 一个实现了缓冲协议的对象，将源对象中的字节序列复制到新建的二进制序列中。

使用缓冲类对象构建二进制序列是一种低层操作，可能涉及类型转换。

```
# 例4-3: 使用数组中的原始数据初始化 bytes 对象
import array
numbers = array.array('h', [-2, -1, 0, 1, 2])
octets = bytes(numbers) # b'\xfe\xff\xff\xff\x00\x00\x01\x00\x02\x00'
```

结构体和内存视图

`struct` 模块提供了一些函数，把打包的字节序列转换成不同类型字段组成的元组，还有一些函数用于执行反向转换，把元组转换成打包的字节序列。`struct` 模块能处理 `bytes`，`bytearray`，`memoryview` 对象。

4.3 基本的编解码器

Python 自带了超 100 中编解码器，用于在文本和字节之间相互转换。每个编解码器都有一个名称，有的有几个别名，比如 ‘`utf_8`’ 别名有 ‘`utf8`’,‘`utf-8`’,‘`U8`’。这些名称可以传给一些函数的 `encoding` 参数。

4.4 了解编解码问题

编解码有个一般性的 UnicodeError 异常，报告错误时几乎都会指明具体的异常

- UnicodeEncodeError：编码错误，在将字符串转换为二进制序列时错误。
- UnicodeDecodeError：解码错误，在将二进制序列转换为字符串时错误。
- SyntaxError：语法错误。

4.4.1 处理 UnicodeEncodeError

多数非 UTF 编解码器只能处理 Unicode 字符的一小部分子集。把文本转换成字节序列时，如果目标编码中没有定义某个字符，那就会抛出 `UnicodeEncodeError` 异常，除非把 `errors` 参数传给编码方法或函数，对错误进行特殊处理。

4.4.2 处理 UnicodeDecodeError

不是每一个字节都包含有效的 ASCII 字符或 UTF 码。因此，将二进制转换为文本时，如果假设是这两个编码中的一个，遇到无法转换的字节序列时会抛出 `UnicodeError`。

此外，很多陈旧的 8 位编码，如 'cp1252'，能解码任何字节序列流而不抛出错误，例如随机噪声。这会悄无声息得得到一串无用的输出。

4.4.3 抛出 SyntaxError

Python 3 默认使用 UTF-8 编码源码，Python 2 则默认使用 ASCII。如果加载得.py 模块中包含 UTF-8 之外得数据，而且没有声明编码，会得到 SyntaxError 错误。

为了修复这类问题，可以在 Python 脚本文件顶部天界一个 `coding` 注释：

```
# coding:cp1252
```

现在 Python3 使用 UTF-8 编码，若要修正源码得陈旧编码问题，最好将其转换成 UTF-8 编码，而不是使用 `coding` 注释。

值得一提的是，Python3 原则上是允许在源码中使用非 ASCII 名称的，原书作者认为这有利有弊。以普遍理性而言，不要这样做。

4.4.4 如何找出字节序列的编码

除非有人告诉你字节序列的编码，不然不能找到正确的编码方式。

仅能根据某个字节码出现的频率猜测字节序列的编码，Python 有一个库：Chardet 可以对文件使用这种方式猜测编码方式。

4.4.5 BOM: 有用的鬼符

观察下面例子：

```
'El'.encode('utf_16') # b'\xff\xfeE\x00l\x00'
```

可以发现使用 `utf_16` 编码的序列开头有几个格外的字节 `b'\xff \xfe'`。这是 BOM，即字节序标记 (byte-order mark)，指明编码时使用 Intel CPU 的小字节序。在大字节序 CPU 中，编码顺序相反。

UTF-16 有两个变种：UTF-16LE 显示指明使用小写字符，UTF-16BE，显式指明使用大写字符。如果使用这两个变种，则不会生成 BOM[18]。

[18]BOM 的概念多为一些协议规则，请根据实际考虑，这里不多做解释。

4.5 处理文本文件

处理文本一般采用“三明治”方法，即在读入文件时解码输入的字符串，在输出文件时将字符串编码成字节序列。而在处理过程中，只处理字符文本。

Python3 便是采用的这种方法，内置的 open 函数会在读取文件时作必要的解码，在写入文件时进行编码，这样我们就只需要关心字符串的处理[19]。

可以见得，处理文本文件十分简单。但是依赖默认编码，往往会遇到问题。Python3 在对文件进行读写时，会默认使用操作系统的编码方式，例如国人用的 Window10 系统默认编码为 GBK，这时候如果读取一个 UTF-8 编码的方式则会出现乱码。为了解决这类问题，在涉及到编码解码的函数中都应该传入 encoding 参数。

如果没有设置 encoding 的值，默认值由 `local.getpreferredencoding()` 提供。

4.6 为了正确比较而规范化 Unicode 字符串

Unicode 有组合字符 (变音符号和附加到前一个字符上的记号，打印时作为一个整体)，处理起来比较复杂。

```
s1 = 'café'
s2 = 'cafe\u0301'
s1 == s2          # False
```

上述代码中 s1 和 s2 表达的字符是一致的，在 Unicode 标准中，‘é’ 和 ‘e\u0303’ 这样的序列叫做 ‘标准等价物’，应用程序应该把它们视作相同的字符。但是 Python 看到的是码位的序列，因此判定二者不同。

这个问题的解决方案是使用 `unicodedata.normalize` 函数提供的 Unicode 规范化。这个函数的第一个参数是四个字符串中的一个，最常用的两个如下：

- NFC(Normalization Form C)
  使用最少的码位构成等价的字符串。是 W3C 推荐的规范化形式。
- NFD
  把组合字符分解成基字符和单独的组合字符。

西式键盘默认是 NFC 形式，但安全起见最好使用 `normalize('NFC',user_text)` 清洗字符串。

还有两个并不常用，一般只在特殊情况使用，这里仅列出不做解释：

- NFKC
  NFC 基础上的严格的规范化格式，多用在一些考虑兼容性的特殊字符处理场合。
- NFKD
  NFD 基础上的严格的规范化格式，多用在一些考虑兼容性的特殊字符处理场合。

[19]Python2.7 用户需要使用 io.open() 函数才能在读写文件时自动解码编码

4.6.1 大小写折叠

大小写折叠其实就是把所有文本变成小写，再做转换。这个功能由 `str.casefold()` 方法支持。

str.casefold() 方法与 str.lower() 方法处理结果几乎一样，自 Python3.4 起，它们处理得到不同结果的有 116 个码位，而 Unicode 6.3 则有 110122 个字符，占比 0.11%，处为特殊情况，否则无需在意。

4.6.2 规范化文本匹配实用函数

对大多数应用来说,NFC 是最好的规范化形式。不区分大小写的比较应该使用 str.casefold()。如果要处理多语言文本，应该有 `nfc_equal` 和 `fold_equal` 函数。

```
# 例4-13: 比较规范化 Unicode 字符串

from unicodedata import normalize

def nfc_equal(str1,str2):
    return normalize('NFC',str1) == normalize('NFC',str2)

def fold_equal(str1,str2):
    return (normalize('NFC',str1).casefold() == normalize('NFC',str2).casefold())

s1 = 'café'
s2 = 'cafe\u0301'

print(s1 == s2)          # False
print(nfc_equal(s1,s2))  # True
print(nfc_equal('A','a')) # False
print(fold_equal('A','a')) # True
```

4.6.3 极端 “规范化”：去掉变音符号

去掉重音符并不是正确的规范化方法，但现实生活往往需要这样做。

```
# 例4-14: 去掉全部组合记号的函数

import unicodedata
import string

def shave_marks(txt):
    """去掉重音符号"""
    norm_txt = unicodedata.normalize('NFD',txt) # 将所有字符分解基字符与组合记号
    shaved = ''.join(c for c in norm_txt if not unicodedata.combining(c)) # 过滤组合记号
    return unicodedata.normalize('NFC',shaved) # 重组所有字符
```

原文还有一些特殊字符的处理方法，但是本人并不会德语/俄文，作为中文用户也用不到这些操作，故省略，这些内容可参阅原书 104 页。

III 把函数视作对象

在 Python 中，函数是一等对象。编程语言理论家定义一等对象如下：

- 在运行时创建
- 能赋值给变量或数据结构中的元素
- 能作为参数传给函数
- 能作为函数的返回结果

5 一等函数

5.1 把函数视作对象

观察下面这个求阶乘的函数：

```
# 例5-1: 阶乘函数
def factorial(n):
    """return n!"""
    return n if n<2 else n * factorial(n-1)

print(factorial(10))      # 3628800
print(factorial.__doc__)  # return n!
print(type(factorial))    # <class 'function'>
```

其中，`__doc__` 属性用于生成对象的帮助文本。从 `type()` 函数的返回值可知，我们创建的 `factorial()` 函数是 `function` 类的实例。

下面例子展现了函数对象的“一等”本性。我们可以把 `factorial` 函数赋值给变量 `fact`，然后通过变量名调用。我们还能把它作为参数传给 `map` 函数。`map` 函数返回一个可迭代对象，里面的元素是把第一个参数 (一个函数) 应用到第二个参数 (一个可迭代对象) 中各个元素上得到的结果。

```
# 例5-2: 函数名参数传递
def factorial(n):
    """return n!"""
    return n if n<2 else n * factorial(n-1)

fact = factorial
print(fact(5))                      # 120
print(list(map(factorial,range(11)))) # [0, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880,
    3628800]
```

5.2 高阶函数

接受函数作为参数，或者把函数作为结果返回的函数是高阶函数。map 函数就是一个例子，此外内置的 sorted 函数也是，可选的 key 参数用于提供一个函数，应用到各个元素上进行排序。任何单参数函数都能作为 key 参数的值。

```
# 例5-3: sorted 函数与 key 参数
fruits = ['strawberry','fig','apple','cherry']
print(sorted(fruits,key=len)) # ['fig', 'apple', 'cherry', 'strawberry']
```

在函数式编程规范中，最为人知的高阶函数有 map,filter,reduce。

map,filter,reduce 的现代替代品

在 Python 3 中 map 和 filter 还是内置函数，使用列表推导和生成器表达式可以在绝大多数情况下替代这两个高阶函数，而且可读性更高。

```
# 例5-5: map filter 与列表推导
def factorial(n):
    """return n!"""
    return n if n<2 else n * factorial(n-1)

fact = factorial

print(list(map(fact,range(6))))  # [0, 1, 2, 6, 24, 120]
print([fact(n) for n in range(6)]) # [0, 1, 2, 6, 24, 120]

print(list(map(fact,filter(lambda n:n % 2,range(6))))) # [1, 6, 120]
print([fact(n) for n in range(6) if n % 2])          # [1, 6, 120]
```

我们可以看到让表达式复杂后，使用列表推导可读性高了许多，并避免了 lambda 表达式的使用。

高阶函数 reduce 在 Python 2 中是内置函数，在 Python3 中被移到了 functools 模块中，多数情况下更加推荐使用 sum 函数。

```
# 例5-6: 使用 reduce 和 sum 求和
from functools import reduce
from operator import add

print(reduce(add,range(99))) # 4851
print(sum(range(99)))        # 4851
```

sum 和 reduce 的通常思想是把某个操作连读应用到序列的元素上，累计之前的结果，把一个系列值归约成一个值。与之类似的还有 all 和 any 两个内置归约函数。

5.3 匿名函数

lambda 关键字在 Python 表达式内创建匿名函数。由于 Python 句法非常简单，导致 lambda 函数的定义体只能使用纯表达式。

在参数列表中最适合使用匿名函数。除此以外，不建议写复杂的 lambda 表达式，因为要么 Python 写不出来，就算写出来了，可读性也很差。

```
# 例5-7: lambda 表达式
fruits = ['strawberry','apple','banana','cherry','fig']
new_fruit = sorted(fruits,key = lambda word:word[::-1])
print(new_fruit)  # ['banana', 'apple', 'fig', 'strawberry', 'cherry']
```

lambda 表达式和 def 函数一样会创建函数对象。

5.4 可调用对象

除了用户定义的函数，调用运算符 () 还可以应用到其他对象上。如果想判断对象能否调用，可以使用内置的 callable() 函数。Python 数据模型文档列出了 7 种可调用对象。

- 用户定义的函数

使用 def 语句或 lambda 表达式创建。

- 内置函数
- 内置方法
- 方法

在类的定义体中定义的函数。

- 类

调用类时会运行类的 __new__ 方法创建一个实例，然后运行 __init__ 方法，初始化实例，最后把实例返回给调用方。因为 Python 没有 new 运算符，所以调用类相当于调用函数。

- 类的实例

如果类定义了 __call__ 方法，那么它的实例可以作为函数调用。下一节将说明。

- 生成器函数

使用 yield 关键字的函数或方法。调用生成器函数返回的是生成器对象。生成器函数在很多方面与其他可调用对象不同，后文将说明。

5.5 用户定义的可调用类型

任何 Python 对象都可以表现得像函数。为此，只需实现实例方法 __call__。

```
# 例5-8: 调用 BingoCage 实例，从打乱的列表中取出一个元素
import random

```

```
class BingoCage:
    def __init__(self,items) -> None:
        self._items = list(items)
        random.shuffle(self._items)

    def pick(self):
        try:
            return self._items.pop()
        except IndexError:
            raise LookupError('pick from empty BingoCage')

    def __call__(self):   # bingo.pick() 的快捷方式是 bingo()
        return self.pick()

bingo = BingoCage(range(3))
print(bingo())           # 2
```

实现 `__call__` 方法的类是创建函数类对象的简便方式，此时必须在内部维护一个状态，让它在调用之前可用。

5.6 函数内省

除了 `__doc__`，函数对象还有很多属性。使用 `dir` 函数可以探知某个函数对象所具备的属性。

先讨论 `__dict__` 内置函数。与用户定义的常规类一样，函数使用 `__dict__` 属性存储赋予它的用户属性。这相当于一种基本形式的注解。

下面重点说明函数专有而用户定义的一般对象没有的属性。计算两个属性集合的差便可获得函数专有的属性列表。

```
# 例5-9: 列出常规对象没有而函数有的属性
class C: pass
def func(): pass
obj = C()
print(sorted(set(dir(func)) - set(dir(obj))))
# ['__annotations__', '__call__', '__closure__', '__code__', '__defaults__', '__get__',
    '__globals__', '__kwdefaults__', '__name__', '__qualname__']
```

这些内置函数的意义如下：

表 3.1 函数特有的内置函数

名称	类型	说明
__annotations__	dict	参数和返回值的注解
__call__	method-wrapper	实现 () 运算符，即可调用对象协议
__closure__	tuple	函数闭包，即自由变量的绑定
__code__	code	编译成字节码的函数元数据和函数定义体
__defaults__	tuple	形式参数的默认值
__get__	method-wrapper	实现只读描述符协议
__globals__	dict	函数所在模块中的全局变量
__kwddefaults__	dict	仅限关键字形式参数的默认值
__name__	str	函数名
__qualname__	str	函数的限定名称

5.7 从定位参数到仅限关键字参数

Python 最好的特征之一是提供了极为灵活的参数处理机制，而且 Python3 进一步提供了仅限关键字参数。与之密切相关的是，调用函数时使用 * 和 ** “展开” 可迭代对象，映射到单个参数。

下面介绍 Python 的四种参数类型[1]。

- 默认参数

默认参数，注意一点：必选参数在前，默认参数在后，否则 Python 的解释器会报错。

- 可变参数

可变参数，意思就是传入参数的个数是可变的，可以是 1 个，2 个，无数个；传入参数类型为 list 或者 tuple。如果已经存在一个数组了，如 i = [1,2,3]，传参的时候前面加上 * 就行，*i 表示把 i 这个数组所有元素作为可变参数传进去。

```
def calc(*numbers):
    sum=0
    for n in numbers:
        sum= sum+n*n
    return sum

calc(1,2,3) # 14
calc(*[1,2,3]) # 14
```

- 关键字参数

关键字参数允许你传入 0 个或任意个含参数名的参数，0 意味着关键字参数可填可不填，这些关键字参数在函数内部自动组装为一个 dict。关键字参数需要在参数名前加 **。同样，如果需要传入字典，传参的时候前面要加上 **。

[1]这些内容为本人添加，原书并没有。

```python
def student(name,age,**interest):
    print('name:',name,' age:',age,' interest:',interest)

student('wang',21,sports='football')
# name: wang age:21 interest:{'sports':'football'}
```

- 仅限关键字参数

关键字参数，对于传入的参数名无法限制。如果想对参数名有限制，就用到了仅限关键字参数。仅限关键字参数需要一个特殊分隔符 *，* 后面的参数被视为命名关键字参数。

```python
def f(a,*,b):
    return a,b

f(1,b=2)   # (1,2)
```

下面给出书上一个例子：

```python
# 例5-10: tag 函数
def tag(name,*content,cls=None,**attrs):
    """生成一个或多个 HTML 标签"""
    if cls is not None:
        attrs['class'] = cls
    if attrs:
        attr_str = ''.join(' %s ="%s"' % (attr,value) for attr, value in
            sorted(attrs.items()))
    else:
        attr_str = ''
    if content:
        return '\n'.join('<%s%s> %s </%s>' % (name,attr_str,c,name) for c in content)
    else:
        return '<%s%s />' % (name,attr_str)

print(tag('br'))                        # <br />
print(tag('p','hello'))                 # <p> hello </p>
print(tag('p','hello',id=33))           # <p id ="33"> hello </p>
print(tag(content='testing',name='img'))
# <img content ="testing" /> 此时的 content 传入 **attrs

my_tag = {'name':'img','title':'Sunset Boulevard','src':'sunset.jpg','cls':'framed'}
print(tag(**my_tag))
# <img class ="framed" src ="sunset.jpg" title ="Sunset Boulevard" />
```

5.8 获取关于参数的信息

函数对象有个 `__defaults__` 属性，它的值是一个元组，里面保存着定位参数和关键字参数的默认值。仅限关键字参数的默认值在 `__kwddefaults__` 属性中。参数的名称在 `__code__` 属性中，它的值是一个 code 对象引用，自身也有很多属性。

```python
# 例5-15:  提取关于函数的信息
```

```
def clip(text,max_len=80):
    """在 max_len 前面或后面的第一个空格处截断文本"""
    end = None
    if len(text) > max_len:
        space_before = text.rfind(' ',0,max_len)
        if space_before >= 0:
            end = space_before
        else:
            space_after = text.rfind(' ',max_len)
        if space_after >= 0:
            end = space_after
    if end is None:
        end = len(text)
    return text[:end].rstrip()

print(clip.__defaults__) # (80,)
print(clip.__code__)
# <code object clip at 0x000002109AE7AAE0, file "<string>", line 2>
print(clip.__code__.co_varnames)
# ('text', 'max_len', 'end', 'space_before', 'space_after')
print(clip.__code__.co_argcount) # 2
```

通过 code 的确能查看一些函数的信息，但是这不是很便利。使用 inspect 模块将更方便。

```
# 例5-17: 提取函数签名
from inspect import signature

def clip(text,max_len=80):
    """在 max_len 前面或后面的第一个空格处截断文本"""
    end = None
    if len(text) > max_len:
        space_before = text.rfind(' ',0,max_len)
        if space_before >= 0:
            end = space_before
        else:
            space_after = text.rfind(' ',max_len)
        if space_after >= 0:
            end = space_after
    if end is None:
        end = len(text)
    return text[:end].rstrip()

sig = signature(clip)
print(sig)        # (text, max_len=80)
for name,param in sig.parameters.items():
    print(param.kind,':',name,'=',param.default)
# POSITIONAL_OR_KEYWORD : text = <class 'inspect._empty'>
# POSITIONAL_OR_KEYWORD : max_len = 80
```

这样就方便许多，inspect.signature 函数返回一个 inspect.Signature 对象，他有一个 parameters 属性，是一个有序映射，把参数名和 inspect.Parameters 对象对应起来。各

个 Parameter 属性也有自己的属性，例如 name,default 和 kind。特殊的 inspect._empty 值表示没有默认值，考虑到 None 是有效的默认值。

kind 属性的值是 _ParameterKind 类中的 5 个值之一:

- POSITIONAL_OR_KEYWORD

可以通过定位参数和关键字参数传入的形参。

- VAR_POSITIONAL

定位参数元组。

- VAR_KEYWORD

关键字参数字典。

- KEYWORD_ONLY

仅限关键字参数。

- POSITIONAL_ONLY

仅限定位参数。

除此以外，inspect.Parameter 对象还有一个 annotation 属性，可能包含 Python3 新的注解句法提供的函数签名元数据。

5.9 函数注解

Python3 提供了一种句法，用于为函数声明中的参数和返回值附加元数据。

```
# 例5-19:  有注解的 clip 函数
def clip(text:str,max_len:'int > 0'=80) -> str:
    """在 max_len 前面或后面的第一个空格处截断文本"""
    end = None
    if len(text) > max_len:
        space_before = text.rfind(' ',0,max_len)
        if space_before >= 0:
            end = space_before
        else:
            space_after = text.rfind(' ',max_len)
        if space_after >= 0:
            end = space_after
    if end is None:
        end = len(text)
    return text[:end].rstrip()
```

函数声明中的各个参数可以在: 之后增加注解表达式。如果参数有默认值，注解放在参数名和 = 号之间。如果想注解返回值，在) 和函数声明末尾的: 之间添加一个 -> 和一个表达式。表达式可以是任何类型，可以是最常用的类 (str,int 等)，也可以是字符串 (如：'int > 0')。

注解不会做任何处理，只是存储在函数的 __annotations__ 属性中：

```
clip.__annotations__
# {'text':<class 'str'>, 'max_len':'int>0', 'return':<class 'str'>}
```

Python 对注解做的唯一事情就是存储在 `__annotations__` 属性中。仅此而已，注解对 Python 解释器没有任何意义。

5.10 支持函数式编程的包

虽然 Python 的目标不是变成函数式编程语言，但通过 `operator` 和 `functools` 等包的支持，函数式编程风格也可以信手拈来。

5.10.1 operator 模块

在函数式编程中，常常需要把算数运算符当作函数使用。其中的一个解决方法是使用 `lambda` 匿名函数。

```
# 例5-21: 使用 reduce 和匿名函数计算阶乘
from functools import reduce

def fact(n):
    return reduce(lambda x,y:x*y , range(1,n+1))
```

`operator` 模块提供了多个算数运算符从而避免了编写 `lambda` 表达式这样平凡的匿名函数。

```
# 例5-22: 使用 reduce 和 operator.mul 计算阶乘
from functools import reduce
from operator import mul

def fact(n):
    return reduce(mul,range(1,n+1))
```

operator 模块中还有一类函数，能替代从序列中取出元素或读取对象属性的 `lambda` 表达式。下面这个例子中，根据元组的某个字段给元组列表排序。`itemgetter(1)` 等效于 `lambda item:item[1]`。

```
# 例5-23: 使用 itemgetter 排序一个元组列表
from operator import itemgetter

metro_data = [
    ('Tokyo', 'JP', 36.9, (35.6, 139.69)),
    ('Delhi NCR', 'IN', 21.9, (28.6, 77.2)),
    ('Mexico City', 'MX', 20.1, (19.4, -99.1)),
    ('New York-Newark', 'US', 20.1, (40.8, -74.0)),
]

for city in sorted(metro_data,key = itemgetter(1)):
    print(city)
# ('Delhi NCR', 'IN', 21.9, (28.6, 77.2))
# ('Tokyo', 'JP', 36.9, (35.6, 139.69))
# ('Mexico City', 'MX', 20.1, (19.4, -99.1))
```

```
# ('New York-Newark', 'US', 20.1, (40.8, -74.0))
```

如果把多个参数传给 itemgetter，它构建的函数会返回提取的值构成的元组。

```
cc_name = itemgetter(1,0)
for city in metro_data:
    print(cc_name(city))

# ('JP','Tokyo')
# ......
```

itemgetter 使用 [] 运算符，因此它支持映射和任何实现 __getitem__ 方法的类。

attrgetter 与 itemgetter 作用类似，它创建的函数根据名称提取对象的属性。如果把多个属性名传给 attrgetter，它也会返回提取的值构成的元组。此外，如果参数名中包括.(点号)，attrgetter 会深入嵌套对象，获取指定的属性。

```
# 例5-24: attrgetter 处理
from collections import namedtuple
from operator import attrgetter

metro_data = [
    ('Tokyo', 'JP', 36.9, (35.6, 139.69)),
    ('Delhi NCR', 'IN', 21.9, (28.6, 77.2)),
    ('Mexico City', 'MX', 20.1, (19.4, -99.1)),
    ('New York-Newark', 'US', 20.1, (40.8, -74.0)),
]

LatLong = namedtuple('Latlong','lat long')
Metropolis = namedtuple('Metropolis','name cc pop coord')
metro_areas = [Metropolis(name,cc,pop,LatLong(lat,long)) for name,cc,pop,(lat,long) in
    metro_data]

name_lat = attrgetter('name','coord.lat')
for city in sorted(metro_areas,key=attrgetter('coord.lat')):
    print(name_lat(city))
# ('Mexico City', 19.4)
# ('Delhi NCR', 28.6)
# ('Tokyo', 35.6)
# ('New York-Newark', 40.8)
```

使用以下代码可以查看 operator 模块中的部分函数：

```
[name for name in dir(operator) if not name.startswith('_')]
# ['abs', 'add', 'and_', 'attrgetter', 'concat', 'contains', 'countOf', 'delitem', 'eq',
    'floordiv', 'ge', 'getitem', 'gt', 'iadd', 'iand', 'iconcat', 'ifloordiv', 'ilshift',
    'imatmul', 'imod', 'imul', 'index', 'indexOf', 'inv', 'invert', 'ior', 'ipow',
    'irshift', 'is_', 'is_not', 'isub', 'itemgetter', 'itruediv', 'ixor', 'le',
    'length_hint', 'lshift', 'lt', 'matmul', 'methodcaller', 'mod', 'mul', 'ne', 'neg',
    'not_', 'or_', 'pos', 'pow', 'rshift', 'setitem', 'sub', 'truediv', 'truth', 'xor']
```

这其中大部分函数作用不言而喻，以 i 开头，后面是另一个运算符的那些名称对应的是

增量运赋值运算符。如果第一个参数是可变的，那么这些运算符函数就会修改它；否则，作用就与不带 `i` 的函数一样。

最后介绍一个 `methodcaller`，它会自行创建函数。`methodcaller` 创建的函数会在对象上调用参数指定的方法。

```
# 例5-25: methodcaller 使用示范
from operator import methodcaller

s = 'The time has come'
upcase = methodcaller('upper')
print(upcase(s))      # THE TIME HAS COME

hiphenate = methodcaller('replace',' ','-')
print(hiphenate(s))   # The-time-has-come
```

5.10.2 使用 `functools.partial` 冻结参数

`functools` 提供了一系列高级函数，除 reduce 外，最有用的是 `partial` 以及其变体 `partialmethod`。

`functools.partial` 这个高阶函数用于部分应用一个函数。部分应用指的是，基于一个函数创建一个新的可调用对象，把原函数的某些参数固定。使用这个函数可以把接受一个或多个参数的函数改编成需要回调的 API，这样参数更少。

```
# 例5-16: partial 函数的使用
from operator import mul
from functools import partial

triple = partial(mul,3)  # 使用 mul 创建 triple 函数，把第一个定位参数定为 3
triple(7)                # 21
print(list(map(triple,range(10)))) # [0, 3, 6, 9, 12, 15, 18, 21, 24, 27]
```

`partial` 的第一个参数是一个可调用对象，后面跟着任意个要绑定的定位参数和关键字参数。

6 使用一等函数实现设计模式

6.1 案例分析：重构 “策略” 模式

如果合理利用作为一等对象的函数，某些设计模式可以简化。

6.1.1 经典的 “策略” 模式

定义一系列算法，把它们一一封装起来，并且使它们可以相互替换。本模式使得算法可以独立于使用它的客户而变化。

——« 设计模式：可复用面向对象软件的基础 »

图 6.1 中的 UML 类图指出的 “策略” 模式对类的编排。

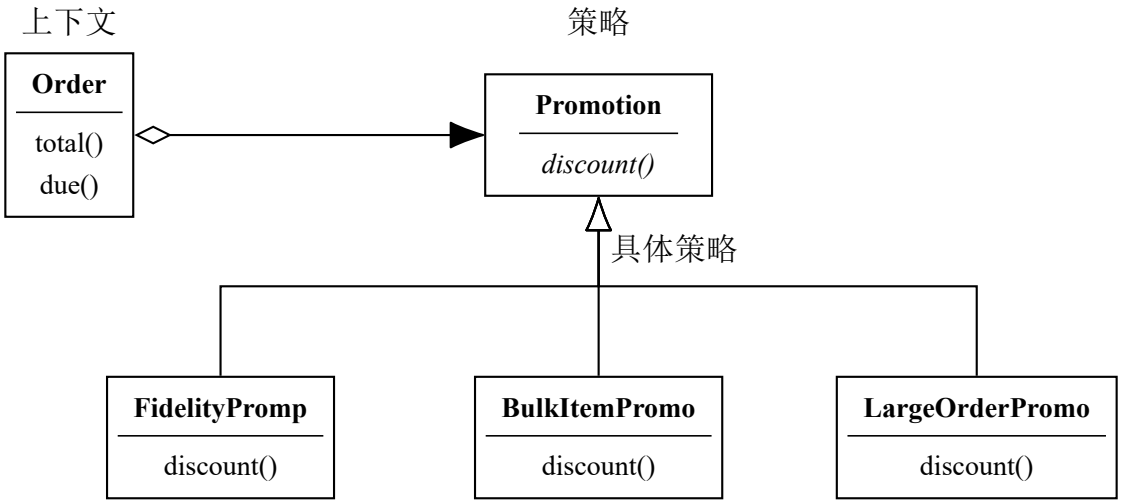

图 6.1 “策略” 设计模式

电商领域有个功能可以明显使用 “策略” 模式，即根据客户的属性或订单中的商品计算折扣。假定一个网店有以下折扣：

- 有 1000 积分或以上顾客，每个折扣享 5% 折扣。
- 同一个订单中，单个商品数量达到 20 个或以上，享 10% 折扣。
- 订单中不同商品达到 10 个或以上，享 7% 折扣。
- 一个订单仅能享受一个折扣

结合图 6.1，各个组成部分意义如下：

- 上下文

把一些计算委托给实现不同算法的可互换组件，它提供服务。在这个例子中，上下文是 `Order`，它会根据不同的算法计算促销折扣。

- 策略

实现不同算法的组件共同的接口。

- 具体策略

“策略”的具体子类。

例 6-1 实现了这个电商折扣中的方案。在这个示例中，实例化订单之前，系统会以某种方法选择一种促销策略，然后把它传给 `Order` 构造方法。具体怎么选策略，不在这个模式的职责范围内。

```python
# 例6-1: 实现 Order 类，支持插入式折扣策略
from abc import ABC, abstractclassmethod, abstractmethod
from collections import namedtuple

Customer = namedtuple('Customer', 'name fidelity')


class LineItem:
    '''购买商品类'''

    def __init__(self, product, quantity, price):
        '''构造方法
        :param product:商品
        :param quantity:商品数量
        :param price:商品价格
        '''
        self.product = product
        self.quantity = quantity
        self.price = price

    def total(self):
        '''
        :return:商品总价
        '''
        return self.price * self.quantity


class Order:
    '''订购类'''

    def __init__(self, customer: Customer, cart: LineItem, promotion=None):
        '''构造方法
        :param customer:用户
        :param cart:购物车
        :param promotion:打折方式(函数)
        '''
        self.customer = customer
        self.cart = cart
        self.promotion = promotion

    def total(self):
        '''计算购物车中商品总价'''
        if not hasattr(self, '__total'):
            self.__total = sum(item.total() for item in self.cart)
        return self.__total
```

```
46
47      def due(self):
48          '''最终价格：总价 - 折扣价'''
49          if self.promotion is None:
50              discount = 0
51          else:
52              discount = self.promotion.discount(self)
53          return self.total() - discount
54
55      def __repr__(self):
56          '''格式化输出'''
57          fmt = '<Order total:{:.2f} dur:{:.2f}>'
58          return fmt.format(self.total(), self.due())
59
60
61  class Promotion(ABC):
62      @abstractmethod
63      def discount(self, order):
64          """返回折扣金额(正值)"""
65
66
67  class FidelityPromo(Promotion):
68      """积分 > 1000 客户，提供 5% 折扣"""
69
70      def discount(self, order):
71          return order.total() * .05 if order.customer.fidelity >= 1000 else 0
72
73
74  class BulkItemPromo(Promotion):
75      """单个商品为 20 或以上，提供 10% 折扣"""
76
77      def discount(self, order):
78          discount = 0
79          for item in order.cart:
80              if item.quantity >= 20:
81                  discount += item.total() * 0.1
82          return discount
83
84
85  class LargeOrderPromo(Promotion):
86      """订单中不同商品达到 10 个或以上，提供 7% 折扣"""
87
88      def discount(self, order):
89          distinct_items = {item.product for item in order.cart}
90          if len(distinct_items) >= 10:
91              return order.total() * .07
92          return 0
93
94
95  joe = Customer('John Doe',0)
96  ann = Customer('Ann Smith',1100)
97  cart = [LineItem('banana',4,.5),
98          LineItem('apple',10,1.5),
```

```
        LineItem('watermelon',5,5.0)]
print(Order(joe,cart,FidelityPromo())) # <Order total:42.00 dur:42.00>
print(Order(ann,cart,FidelityPromo())) # <Order total:42.00 dur:39.90>

banana_cart = [LineItem('banana',30,.5),
               LineItem('apple',10,1.5)]
print(Order(joe,banana_cart,BulkItemPromo())) # <Order total:30.00 dur:28.50>

long_order = [LineItem(str(item_code),1,1.0)
              for item_code in range(10)]
print(Order(joe,long_order,LargeOrderPromo())) # <Order total:10.00 dur:9.30>
```

在上面这个例子中，把 `Promotion` 定义为抽象基类 (ABC)，这么做是为了使用 @abstract-method 修饰器，从而明确表明所用的模式。

6.1.2 使用函数实现 “策略” 模式

在实例 6-1 中，每个具体的策略都是一个类，而且只定义了一个方法 `discount`。此外，策略实例没有状态 (即实例属性)。它们看起来更像普通的函数，下面对它们进行重构[2]，把具体的策略换成了普通的函数。

```
# 例6-3: Order 类和使用函数实现的折扣策略
from collections import namedtuple

Customer = namedtuple('Customer', 'name fidelity')


class LineItem:
    '''购买商品类'''

    def __init__(self, product, quantity, price):
        '''构造方法
        :param product:商品
        :param quantity:商品数量
        :param price:商品价格
        '''
        self.product = product
        self.quantity = quantity
        self.price = price

    def total(self):
        '''
        :return:商品总价
        '''
        return self.price * self.quantity


class Order:
    '''订购类'''
```

[2]为减少代码量，不再重复调用实例，仅给出类和函数

```
29
30      def __init__(self, customer: Customer, cart: LineItem, promotion=None):
31          '''构造方法
32          :param customer:用户
33          :param cart:购物车
34          :param promotion:打折方式(函数)
35          '''
36          self.customer = customer
37          self.cart = cart
38          self.promotion = promotion
39
40      def total(self):
41          '''计算购物车中商品总价'''
42          if not hasattr(self, '__total'):
43              self.__total = sum(item.total() for item in self.cart)
44          return self.__total
45
46      def due(self):
47          '''最终价格：总价 - 折扣价'''
48          if self.promotion is None:
49              discount = 0
50          else:
51              discount = self.promotion(self)
52          return self.total() - discount
53
54      def __repr__(self):
55          '''格式化输出'''
56          fmt = '<Order total:{:.2f} dur:{:.2f}>'
57          return fmt.format(self.total(), self.due())
58
59  def fidelity_promo(order):
60      '''积分 > 1000 的顾客：5% 折扣'''
61      return order.total() * 0.05 if order.customer.fidelity >= 1000 else 0
62
63  def bulk_promo(order):
64      '''单个商品为 20 个或以上：10% 折扣'''
65      discount = 0
66      for item in order.cart:
67          if item.quantity >= 20:
68              discount += item.toal() * 0.1
69      return discount
70
71  def large_order_promo(order):
72      '''订单中的不同商品达到 10 个或以上：7% 折扣'''
73      distinct_items = {item.product for item in order.cart}
74      if len(distinct_items) >= 10:
75          return order.total() * 0.07
76      return 0
```

使用参数不仅减少了代码量，而且调用函数比实例化类更简单。

6.1.3 选择最佳策略：简单的方式

下面我们定义一个函数 best_promo 来选者折扣最大的方案。

```
promos = [fidelityPromp,bulk_item_promo,large_order_promo]

def best_promo(order):
    '''选择最佳折扣'''
    return max(promo(order) for promo in promos)
```

在习惯了一等函数后，自然而然会像上述代码一样构建数据结构存储函数，并用生成器表达式使用函数。

这样定义最佳方案选择函数简单且易于阅读，但有些复杂情形下并不适用。比如要加入新的折扣方案，这时候就必须重写 best_promo 函数。

6.1.4 找出模块中的全部策略

在 Python 中，模块也是一等对象，而且标准库提供了几个处理模块的函数。Python 文档是这样说明内置函数 globals 的。

- globals()

  返回一个字典，表示当前的全局符号表。这个符号表始终指向当前模块 (对函数或方法来说，是指定义它们的模块，而不是调用它们的模块)。

下面例子使用 globals 函数帮助 best_promo 自动找到其他可用的 *_promo 函数，过程有点曲折。

```
promos = [globals()[name] for name in globals() # 迭代 globals() 返回字典中各个 name
          if name.endswith('_promo')
          and name != 'best_promo']          # 对各个 name 进行赛选

def best_promo(order):
    '''选择最佳折扣'''
    return max(promo(order) for promo in promos) # 内部代码没有变化
```

收集所有可用促销的另一种方法是,在一个单独的模块中保存所有策略函数,把 best_promo 排除在外。

```
promos = [func for name,func in inspect.getmembers(promotions, inspect.isfunction)]

def best_promo(order):
    '''选择最佳折扣'''
    return max(promo(order) for promo in promos) # 内部代码没有变化
```

在上面代码中，最大的变化是内省名为 promotions 的独立模块，构建策略函数列表。注意要导入 promotions 模块。inspect.getmembers 函数用于获取对象的属性，第二个参数是可选的判断条件 (一个布尔函数)。

不管怎么明明策略函数，上例都可用。唯一重要的是，promotions 模块只能包含计算订

单折扣的函数。如果有人在 promotions 模块中使用不同的签名定义函数，那么 best_promo 函数常识将其应用到订单上时会出错。

动态收集促销折扣函数的一种更为显示的方法是使用装饰器，将在下一节讨论。

6.2 “命令” 模式

“命令” 设计模式将函数作为参数传递而简化。

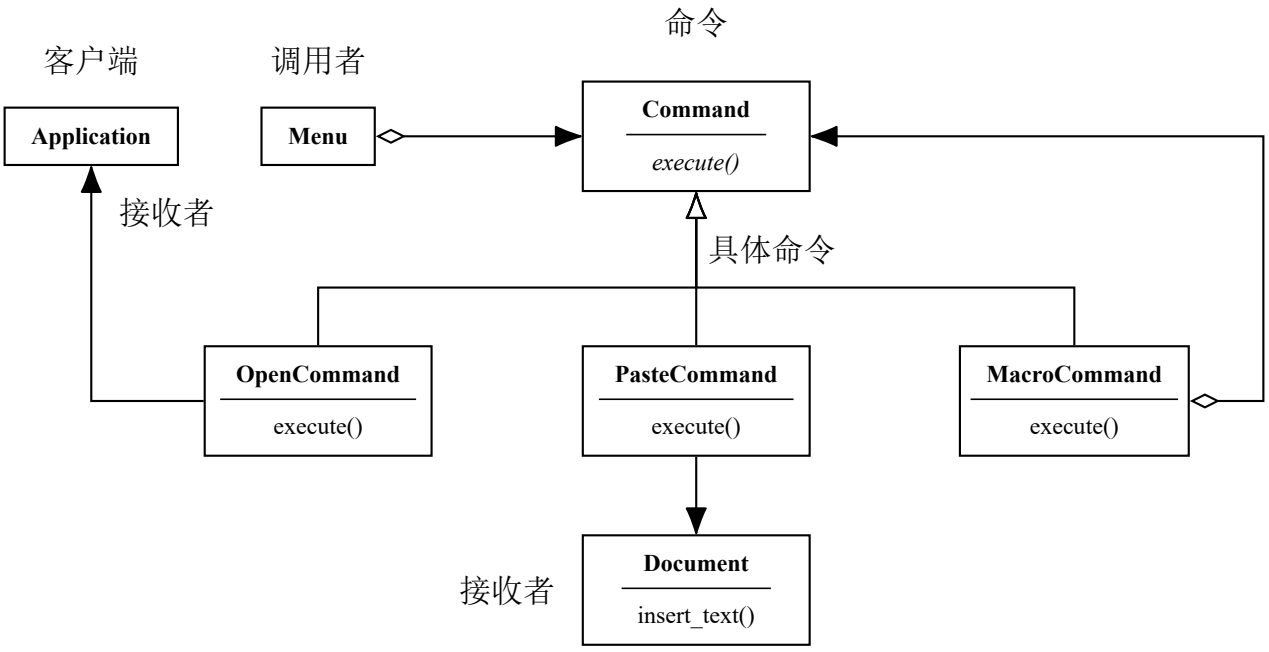

图 6.2 “命令” 设计模式

“命令” 模式的目的是解耦调用操作的对象 (调用者) 和提供实现的对象 (接收者)。这种模式的做法是：在两者之间放一个 Command 对象，让它实现只有一个方法 (execute) 的接口，调用接收者中的方法执行所需的操作。这样，调用者无需了解接收者的接口，而且不同的接收者可以适应不同的 Command 子类。调用者有一个具体的命令，通过调用 execute 方法执行。

我们可以不为调用者提供 Command 实例，而是给它一个函数。此时，调用者不用调用 command.execute() 而是直接调用 command() 即可。MacroCommand 可以实现成定义了 __call__ 方法的类。

```
class MacroCommand:
    '''一个执行一组命令的命令'''
    def __init__(self,commands):
        self.commands = list(commands)

    def __call__(self):
        for command in self.commands:
            command()
```

7 函数装饰器和闭包

函数装饰器用于在源码中 “标记” 函数，以某种方式增强函数的行为。这是一项强大的功能，掌握之前必须理解闭包。

7.1 装饰器基础知识

装饰器是可调用的对象，其参数是另一个函数 (被装饰的函数)。[3]装饰器可能会处理被装饰的函数[4]，然后把它返回，或者将其转换成另一个函数或可调用对象。

假定有个装饰器 `decorate`。

```
@decorate
def target():
    print('running target')
```

等效写法：

```
def target():
    print('running target')

target = decorate(target)
```

这两种写法的效果是一致的：被修饰后的函数执行完毕后得到的 `target` 不一定是原来那个 `target` 函数，而是 `decorate(target)` 返回的函数。

下面看一个装饰器的例子：

```
# 例7-1: 装饰器将一个函数转换成另一个函数
def deco(func):    # 在函数中嵌套定义了函数
    def inner():
        print('running inner()')
    return inner


@deco
def target():
    print('running target')


target()          # 调用被装饰的 target 其实会运行 inner。
# running inner()
print(target)     # 发现 terget 现在是 inner 的引用
# <function deco.<locals>.inner at 0x0000016889199A60>
```

严格来说，装饰器只是语法糖。装饰器可以像常规的可调用对象那样调用，其参数是另一个函数。这样做在元编程中非常方便。

[3]也有类装饰器，在后文讲到。

[4]处不处理取决于修饰器函数内部的具体实现

装饰器的一大特性是能把被装饰的函数转换成其他函数。第二个特性是，装饰器在加载模块时立即执行。

7.2 Python 何时执行装饰器

装饰器的一个关键特性是，他们在被装饰的函数定义之后立即运行。通常是导入模块时。

```
# 例7-2: 装饰器的执行
registry = [] # 保留被修饰的函数的引用


def register(func):
    print('running register(%s)' % func)
    registry.append(func)
    return func # 这里返回的与参数传入的一样，没有进行处理


@register
def f1():
    print('running f1')


@register
def f2():
    print('running f2')


def f3():
    print('running f3')


def main():
    print('running main')
    print('register ->', register)
    f1()
    f2()
    f3()


if __name__ == '__main__': # 只有把该文件当作脚本时才调用 main()
    main()

# running register(<function f1 at 0x000002A1961EEF70>)
# running register(<function f2 at 0x000002A19620D040>)
# running main
# register -> <function register at 0x000002A1961EEEE0 >
# running f1
# running f2
# running f3
```

在上例中可以发现，装饰函数 `register()` 在主函数运行之前运行了两次。如果是导入该模块，则修饰函数同样会被立即调用两次。

考虑到装饰器在真实代码中的常用方式，示例 7-2 有两个不同寻常的地方。

- 装饰器通常在一个单独的模块中定义。
- 大多数装饰器会在内部定义一个函数，然后将其返回。

7.3 使用装饰器改进 “策略” 模式

在电商示例中，遇到的一个问题是当添加新的折扣时，选择最全局佳折扣需要考虑到新添的折扣。下面例子用装饰器解决了这个问题。

```
# 例7-3: register 模块
promos = []

def promotion(promo_func):
    promos.append(promo_func)
    return promo_func

@promotion
def fidelity_promo(order):
    '''积分 > 1000 的顾客: 5% 折扣'''
    return order.total() * 0.05 if order.customer.fidelity >= 1000 else 0

@promotion
def bulk_promo(order):
    '''单个商品为 20 个或以上: 10% 折扣'''
    discount = 0
    for item in order.cart:
        if item.quantity >= 20:
            discount += item.toal() * 0.1
    return discount

@promotion
def large_order_promo(order):
    '''订单中的不同商品达到 10 个或以上: 7% 折扣'''
    distinct_items = {item.product for item in order.cart}
    if len(distinct_items) >= 10:
        return order.total() * 0.07
    return 0

def best_promo(order):
    '''选择可用的最佳折扣'''
    return max(promo(order) for promo in promos)
```

与之前的方案相比，使用装饰器有几个优点：

- @promotion 装饰器突出了被装饰的函数的作用，还便于临时禁用某个促销策略：只把装饰器注释掉。

- 促销折扣策略可以在其他模块中定义，在系统任何地方都行，只需要有装饰器。

不过，多数装饰器会使用内部函数修改被装饰的函数。使用内部函数的代码几乎都要靠闭包才能正确运作。为了理解闭包，需要了解 Python 中的变量作用域。

7.4 变量作用域规则

下面看一个例子：

```
    b = 6
    def f2(a):
    print(a)
    print(b)
    b = 9

    f2(3)  # 3 UnboundLocalError...
```

这里先声明了全局变量 b，但是在函数运行时却出错了。原因是：Python 编译函数的定义体时，它判断 b 是局部变量，因为在函数中给它赋值了。Python 会尝试从本地环境获取 b 。后面调用 f2(3) 时，f2 的定义体会获取并打印局部变量 a 的值。但是尝试获取局部变量 b 的值时，发现 b 没有绑定值。也就是说如果给某个与全局变量同名的局部变量赋值了，那么该局部变量需要在声明后才能正确被调用，而不能反过来使用。

这种情形是设计选择：Python 不要求声明变量，但是假定在函数定义体中赋值的变量是局部变量，无论赋值语句在何处。

如果在函数中赋值时想让 b 作为全局变量，需要使用 global 声明，同时对改变量的修改会影响全局：

```
# 例7-5: global 关键字
b = 6
def f3(a):
    global b
    print(a)
    print(b)
    b = 100

f3(10)     # 10  6
print(b)   # 100
```

7.5 闭包

人们有时会把闭包和匿名函数弄混，这是因为，在函数内部定义函数不常见，直到开始使用匿名函数才会这样做。而且，只有涉及到嵌套函数时才有闭包问题。

闭包指延伸了作用域的函数，其中包括函数定义体中引用，但是不在定义体中定义的非全局变量。函数是不是匿名没有关系，关键是它能访问定义体之外定义的非全局变量。

假如有个名为 avg 的函数，它的作用是计算不断增加的系列值的均值；例如，整个历史中某个商品的平均收盘价。每天都会增加新价格，因此平均值要考虑至目前为止所有的价格。

初学者可能会这样使用类实现：

```
# 例7-8: 计算移动平均值的类
class Averager():

    def __init__(self) -> None:
        self.series = []

    def __call__(self, new_value):
        self.series.append(new_value)
        return sum(self.series)/len(self.series)

avg = Averager()

print(avg(10))    # 10
print(avg(12))    # 11
print(avg(14))    # 12
```

下面使用高阶函数来实现：

```
# 例7-9: 高阶函数计算平均值
def make_averager():
    '''
    计算平均值，需要传一个新值
    '''
    series = []

    def averager(new_value):
        series.append(new_value)
        return sum(series)/len(series)

    return averager

avg = make_averager()
print(avg(10))        # 10
print(avg(12))        # 11
print(avg(14))        # 12
```

调用 make_averager 时，返回一个 averager 函数对象。每次调用 averager 时，它会把参数添加到系列值中，然后计算当前平均值。

在这两个示例中，调用 Average() 或 make_averager() 得到一个可调用对象 avg，它会随历史更新，然后计算当前均值。

Average 类的实例 avg 在哪里存储历史值很明显：self.series 实例属性。但是第二个实例中呢？series 是 make_averager 函数的局部变量，因为那个函数的定义体中初始化了 series 为空列表。可是，调用 avg(10) 时，make_averager 函数已经返回，而它的本地作用域也一去不复返。

在 averager 函数中，series 是自由变量。这是一个技术术语，指未在本地作用域中绑定的变量。

审查返回的 averager 对象，我们发现 Python 在 __code__ 属性中保存局部变量和自由变量的名称。

```
>>> avg.__code__.co_varnames
('new_value','total')
>>> avg.__code__.co_freevars
('series',)
```

series 的绑定在返回在 avg 函数的 __closure__ 属性中。avg.__closure__ 中的各个元素对应于 avg.__code__.co_freevars 中的一个名称。这些元素是 cell 对象，有个 cell_contents 属性，保证着真正的值。

```
>>> avg.__code__.co_freevars
('series',)
>>> avg.__colsure__
(<cell at 0x...: list object at 0x...>,)
>>> avg.__colsure__[0].cell_contents
[10,11,12]
```

综上，闭包是一种函数，它会保留定义函数时存在的自由变量的绑定，这样调用函数时，虽然定义作用域不可用了，但是仍能使用这些绑定。

注意，只有嵌套在其他函数中的函数才可能需要处理不在全局作用域中的外部变量。

7.6 nonlocal 声明

在示例 7-9 中，mark_averager 函数的方法效率并不高。因为我们把所有值存储在历史数列中，然后在每次调用 averager 时使用 sum 求和。更好的方法是，只存储目前的总值和元素个数，然后计算平均值。

```
# 例 7-13: 改进后的平均值函数(有缺陷)
def make_averager():
    total = 0
    count = 0

    def averager(new_value):
        count += 1
        total += new_value
        return total/count

    return averager
```

当我们尝试运行时，会出现如下错误：

```
    >>> avg = make_averager()
    >>> avg(10)
    ...
```

```
    UnboundLocalError: local variable 'count' referenced before assignment
```

问题是，count 是数字或任何不可变类型时，count += 1 语句会赋予 count 新值，这会把 count 变成局部变量。total 变量也受这个影响。

示例 7-9 没有出现这个问题，因为 series 是一个列表，是可变对象。但是对于不可变对象，如果尝试重新绑定，则会隐式地创建局部变量。这样就不再是自由变量了，也不会保存在闭包中。

为了解决这个问题，Python3 引入了 nonlocal 声明，其作用是把变量标记为自由变量，即使在函数中为变量赋予新值了，也会变成自由变量。

7.7 实现一个简单的装饰器

下面定义了一个修饰器，在每次调用被修饰的函数时计时。

```
# 例7-15: 修饰器计算运行时间
import time

def clock(func):
    def clocked(*args):
        t0 = time.perf_counter()
        result = func(*args)      # clocked 的闭包中包含自由变量 func
        elapsed = time.perf_counter() - t0
        name = func.__name__
        arg_str = ','.join(repr(arg) for arg in args)
        print('[%0.8fs] %s(%s) -> %r' % (elapsed,name,arg_str,result))
        return result
    return clocked

@clock
def snooze(seconds):
    time.sleep(seconds)

@clock
def factorial(n):
    return 1 if n<2 else n*factorial(n-1)

snooze(.123)
# [0.12331390s] snooze(0.123) -> None
factorial(5)
# [0.00000040s] factorial(1) -> 1
# [0.00020120s] factorial(2) -> 2
# [0.00039360s] factorial(3) -> 6
# [0.00056890s] factorial(4) -> 24
# [0.00074810s] factorial(5) -> 120
```

这是修饰器的典型行为：把被装饰的函数替换成新函数，二者接受相同的参数，而且 (通常) 返回被装饰的函数本该返回的值。同时还会做些格外操作。

例 7-15 中实现的 clock 装饰器有几个缺点：不支持关键字参数，而且遮盖了被装饰函数的 __name__ 和 __doc__ 属性。下例使用 functools.wraps 装饰器把相关的属性从 func 复制到 clocked 中。此外，这个新版还能正确处理关键字参数。

```
# 例7-17: 改进后的 clock 装饰器
import time
import functools

def clock(func):
    @functools.wraps(func)
    def clocked(*args, **kwargs):
        t0 = time.perf_counter()
        result = func(*args)
        elapsed = time.perf_counter() - t0
        name = func.__name__
        arg_lst = []
        if args:
            arg_lst.append(','.join(repr(arg) for arg in args))
        if kwargs:
            pairs = ['%s=%r' % (k,w) for k,w in sorted(kwargs.items())]
            arg_lst.append(','.join(pairs))
        arg_str = ','.join(arg_lst)
        print('[%0.8fs] %s(%s) -> %r' % (elapsed,name,arg_str,result))
        return result
    return clocked
```

7.8 标准库中的装饰器

Python 内置了三个用于装饰方法的函数：property，classmethod，staticmethod。这将在后面章节做详细说明。另一个常见的装饰器是 functools.wraps，它的作用是协助构建行为良好的修饰器。在例 7-17 中已经用过

此外标准库中最值得关注的是 lru_cache 和 singledispath。这两个装饰器都在 functools 模块中定义。

7.8.1 使用 functools.lru_cache 做备忘

functools.lru_cache 是非常实用的装饰器，它实现了备忘功能，这是一项技术优化，它会把耗时的函数结果保存起来，避免传入同样的参数时重复计算。LRU 三个字母是 “Least Recently Used” 的缩写，表示缓存不会无限增长，一段时间不用的缓存条目会被扔掉。

运行下面例子，获取斐波那契数列：

```
# 例7-18: 生成第 n 个斐波那契数列，递归方法
import functools
import time

def clock(func):
```

```
    @functools.wraps(func)
    def clocked(*args, **kwargs):
        t0 = time.perf_counter()
        result = func(*args)
        elapsed = time.perf_counter() - t0
        name = func.__name__
        arg_lst = []
        if args:
            arg_lst.append(','.join(repr(arg) for arg in args))
        if kwargs:
            pairs = ['%s=%r' % (k,w) for k,w in sorted(kwargs.items())]
            arg_lst.append(','.join(pairs))
        arg_str = ','.join(arg_lst)
        print('[%0.8fs] %s(%s) -> %r' % (elapsed,name,arg_str,result))
        return result
    return clocked

@clock
def fibonacci(n):
    return n if n < 2 else fibonacci(n-1) + fibonacci(n-2)

print(fibonacci(6))
```

在终端运行的结果如下：

```
    [0.00000040s] fibonacci(1) -> 1
    [0.00000080s] fibonacci(0) -> 0
    [0.00050050s] fibonacci(2) -> 1
    [0.00000080s] fibonacci(1) -> 1
    [0.00096630s] fibonacci(3) -> 2
    [0.00000120s] fibonacci(1) -> 1
    [0.00000060s] fibonacci(0) -> 0
    [0.00038900s] fibonacci(2) -> 1
    [0.00166720s] fibonacci(4) -> 3
    [0.00000050s] fibonacci(1) -> 1
    [0.00000070s] fibonacci(0) -> 0
    [0.00029650s] fibonacci(2) -> 1
    [0.00000080s] fibonacci(1) -> 1
    [0.00060670s] fibonacci(3) -> 2
    [0.00255880s] fibonacci(5) -> 5
    [0.00000070s] fibonacci(1) -> 1
    [0.00000070s] fibonacci(0) -> 0
    [0.00034090s] fibonacci(2) -> 1
    [0.00000060s] fibonacci(1) -> 1
    [0.00060460s] fibonacci(3) -> 2
    [0.00000050s] fibonacci(1) -> 1
    [0.00000070s] fibonacci(0) -> 0
    [0.00031580s] fibonacci(2) -> 1
    [0.00113680s] fibonacci(4) -> 3
    [0.00403140s] fibonacci(6) -> 8
    8
```

浪费时间的地方很明显，fibonacci(1) 调用了 8 次，fibonacci(2) 调用了 5 次..... 如果我们采用 lru_cache 则会提高效率。

```
# 例7-19: 生成第 n 个斐波那契数列，lru 修饰改善
import functools
import time

def clock(func):
    @functools.wraps(func)
    def clocked(*args, **kwargs):
        t0 = time.perf_counter()
        result = func(*args)
        elapsed = time.perf_counter() - t0
        name = func.__name__
        arg_lst = []
        if args:
            arg_lst.append(','.join(repr(arg) for arg in args))
        if kwargs:
            pairs = ['%s=%r' % (k,w) for k,w in sorted(kwargs.items())]
            arg_lst.append(','.join(pairs))
        arg_str = ','.join(arg_lst)
        print('[%0.8fs] %s(%s) -> %r' % (elapsed,name,arg_str,result))
        return result
    return clocked

@functools.lru_cache() # 这里加括号的原因后文说明
@clock                 # 叠放了修饰器
def fibonacci(n):
    return n if n < 2 else fibonacci(n-1) + fibonacci(n-2)

print(fibonacci(6))
```

这样一来，执行时间将减半，而且 n 的每个值只调用一次函数：

```
[0.00000020s] fibonacci(1) -> 1
[0.00000050s] fibonacci(0) -> 0
[0.00020570s] fibonacci(2) -> 1
[0.00028600s] fibonacci(3) -> 2
[0.00036260s] fibonacci(4) -> 3
[0.00041440s] fibonacci(5) -> 5
[0.00046940s] fibonacci(6) -> 8
8
```

随着运算量级的增大，lru_cache 修饰器的作用将更为明显。除此以外，lru_cache 在从 Web 中获取信息的应用中也发挥着巨大的作用。

特别需要注意的是，lru_cache 可以使用两个可选的参数来配置，这也是使用他作为修饰时需要加括号的原因：

```
functools.lru_cache(maxsize = 128,typed = False)
```

maxsize 指定存储多少个调用的结果。缓存满了之后，旧的结果会被扔掉。为了保存最佳性能，maxsize 的值应该为 2 的幂。typed 如果设置为 True，则会把不同参数类型得到的

结果分开保存。此外，`lru_cache` 使用字典存储结果，而且键根据调用时传入的定位参数和关键字参数创建，所有被其修饰的函数，它的所有参数都必须是可散列的。

7.8.2 单分派泛函数

因为 Python 不支持重载方法或函数，所以我们不能使用不同的签名定义函数的变体，也无法使用不同的方法处理不同的数据类型。我们可以使用 `if/elif/elif` 调用对应的函数，但是这样不便于模块的用户拓展，还显得笨拙，各个函数之间的耦合也会更紧密。

Python 3.4 新增的 `functools.singledispatch` 装饰器可以把整体方案拆封成多个模块，甚至可以为你无法修改的类提供专门的函数。使用 `@singledispatch` 装饰的普通函数会变成泛函数 (generic function)：根据第一个参数的类型，以不同方式执行相同操作的一组函数。

使用 `@singledispatch` 装饰一个函数，将定义一个泛型函数，获得分派的依据是第一个参数类型。[5]

```
from functools import singledispatch

@singledispatch
def fun(arg,verbose=False):
    if verbose:
        print("Let me just say,",end = " ")
    print(arg)
```

使用泛函数的 `register()` 属性，重载泛函数的实现。泛函数的 `register()` 属性是一个装饰器。对于有类型注释的函数，这个装饰器将自动匹配第一个参数为该类型的已注册函数重载泛函数。

```
@fun.register
def _(arg:int,verbose=False):
    if verbose:
        print("Strength in numbers, eh?", end=" ")
    print(arg)

@fun.register
def _(arg:list,verbose=False):
    if verbose:
        print("Enumerate this:")
    for i,elem in enumerate(arg):
        print(i,elem)
```

此时，当我们调用泛函数 `fun` 时，就能根据第一个参数的类型来分派相应的函数来重载实现。

```
>>> fun(42,verbose=True)
Strength in number, eh? 42
>>> fun(['spam','spam','eggs'] , verbose=True)
Enumerate this:
```

[5]原文为 HTML 相关的示例，不是很直观，这里参考了：https://blog.csdn.net/zjz155/article/details/88593291

```
0 spam
1 spam
2 eggs
3 spam
```

如果函数本身没有类型注释，也可以在 register() 装饰器中声明合适的类型。

```
@fun.register(complex)
def _(arg , verbose=False)
    if verbose:
        print("Better than complicated", end=" ")
    print(arg.real, arg.img)
```

为了能注册之前存在的函数和匿名函数，register() 属性可以当作功能函数使用。

```
def nothing(arg, verbose=False):
print("Nothing.")

fun.register(type(None), nothing)
```

如果泛函数给出的具体类型，没有对应的注册函数的实现，那么泛函数将去寻找更一般化的实现。用 `@singledispatch` 装饰的原函数被注册了基本类型 `object` 类型，也就是说如果找不到更好的实现，那么将使用 `@singledispatch` 装饰的原函数：

只要可能，注册专门函数应该处理抽象基类，不要处理具体实现，这样代码支持的兼容类型更广泛。

`singledispatch` 机制的一个显著特征是，你可以在系统的任何地方和任何模块中注册专门函数。如果后来在新的模块中定义了新的类型，可以轻松地添加一个新的专门函数来处理那个类型。此外，你还可以为不是自己编写地或者不能修改的类型添加自定义函数。

7.9 叠放装饰器

由于修饰器是函数，因此可以组合起来使用。下面两个代码块实现的效果是相同的：

```
@d1
@d2
def f():
    print('f')
```

```
def f():
    print('f')
f = d1(d2(f))
```

7.10 参数化装饰器

解析源码中的装饰器时，Python 把被装饰的函数作为第一个参数传给装饰器函数。如果要让装饰器接受其他参数，则需要创建一个装饰器工厂函数，把参数传给它，返回一个装饰器，然后再把它应用到要装饰的函数上。

下面以我们见过的最简单的装饰器为例：

```
# 例7-22: 示例 7-2 的简化版
registry = []


def register(func):
    print('running register(%s)' % func)
    registry.append(func)
    return func


@register
def f1():
    print('running f1()')


print('running main()')
print('registry ->', registry)
f1()
```

7.10.1 一个参数化的注册装饰器

为了让 `register` 同时具备可选的注册和注销功能，需要设置一个参数，将其设为 `False` 时，不注册被装饰的函数。

```
# 例7-23: 为了接受参数，新的 register 装饰器必须作为函数调用
registry = set()  # 集合添加和删除函数的速度更快


def register(active=True):   # 接受一个可选参数
    def decorate(func):      # decorate 函数是真正的装饰器，它的参数是一个函数
        print('running register(active=%s) -> decorate(%s)' % (active, func))
        if active:
            registry.add(func)
        else:
            registry.discard(func)
        return func          # 装饰器返回函数
    return decorate          # 装饰器工厂函数返回 decorate


@register(active=False)      # 工厂函数必须作为函数调用，并且传参
def f1():
    print('funning f1')

```

```
20
21 @register()                  # 即使不传参，也必须作为函数调用
22 def f2():
23     print('running f2')
```

从概念上看，新的 register 函数不是装饰器而是一个装饰器工厂函数。调用它会返回正在的装饰器。这里的关键是，register() 要返回 decorate，然后把它应用到被装饰的函数上。

如果不使用 @ 句法，则要像常规函数那样使用 register；若想把 f 添加到 registry 中，则装饰 f 的句法是 register()(f)，比如说 register(active=False)(f1)。

参数化装饰器的原理相当复杂。参数化装饰器通常会把被装饰的函数替换掉，而且结构上需要多一层嵌套。接下来讨论这种函数金字塔。

7.10.2 参数化 clock 装饰器

本节继续讨论 clock 修饰器，添加一个功能：让用户传入一个格式字符串，控制被装饰函数的输出：

```
1  # 例7-25: 参数化 clock 装饰器
2  import time
3
4  DEFAULT_FORMAT = '[{elapsed:0.8f}s] {name}({args}) -> {result}'
5
6
7  def clock(fmt=DEFAULT_FORMAT):   # clock 是参数化装饰器的工厂函数
8      def decorate(func):          # decorate 是真正的装饰器
9          def clocked(*_args):     # 包装被修饰的函数
10             t0 = time.time()
11             _result = func(*_args) # 被修饰的函数返回的真正的结果
12             elapsed = time.time() - t0
13             name = func.__name__
14             args = ','.join(repr(arg) for arg in _args) # args 是用于显示的字符串
15             result = repr(_result)
16             # 这里的 **local() 是为了在 fmt 中引用 clocked 的局部变量
17             print(fmt.format(**locals()))
18             return _result
19         return clocked
20     return decorate
21
22
23 @clock()
24 def snooze(seconds):
25     time.sleep(seconds)
26
27
28 for i in range(3):
29     snooze(.123)
30 # [0.12497544s] snooze(0.123) -> None
31 # [0.13129973s] snooze(0.123) -> None
```

```
# [0.13858795s] snooze(0.123) -> None
```

上面的代码比较复杂，看懂就可以解决平时大部分问题了。

也有人[6]认为装饰器最好通过实现 `__call__` 方法的类实现，不应该像本章示例那样通过函数实现。

[6]原书的技术审校

IV 面向对象惯用法

8 对象引用，可变性和垃圾回收

8.1 变量不是盒子

Python 的变量类似于 Java 中的引用式变量，因此最好把它理解为附加在对象上的标注。

```
# 例8-1: 将变量视作标签
a = [1,2,3]
b = a
a.append(4)
print(b)   # [1, 2, 3, 4]
```

在上面例子中，如果将 a,b 理解为单独的盒子的话，在 a 中装入新元素，但查看 b 时却也出现了新元素。如果将列表整体视为一个物品，a,b 仅是在上面的标签，这样更正确，因为 a,b 的本质只是一个指向具体内存地址的指针。

为了理解 Python 中的赋值语句，应该始终先读右边。对象在右边创建或获取，在此之后左边的变量才会绑定到对象上，这就像为对象贴上标注。

8.2 标识，相等性和别名

因为变量只不过是标注，所以无法阻止贴多个标注，标注贴多了，就是别名。

```
# 例8-3: 别名
charles = {'name':'Charles L. Dragon','born':1832}
lewis = charles
print(lewis == charles)  # True
print(id(lewis))         # 1391967345784
print(id(charles))       # 1391967345784
```

如果我们重新创建一个数据相同的字典，并将某个变量分配给它，这个变量就不是上一个变量别名，它们指向不同的内存空间：

```
# 例8-3: 两个相同数据类型的变量
charles = {'name':'Charles L. Dragon','born':1832}
alex = {'name':'Charles L. Dragon','born':1832}

print(charles == alex) # True: 内部数据一样
print(charles is alex) # False: 不是相同对象，所指的内存空间不一样
print(id(charles))    # 1391934254840
print(id(alex))       # 1391934254840
```

关于 id() 函数，在 CPython(最常用的解释器) 中，它会返回对象的内存地址，在其他解释器中可能式别的值，但是关键的是，它的返回值一定是唯一的数值标注，而且在对象的生命周期中绝对不会变。

8.2.1 在 == 和 is 之间选择

== 运算符比较两个对象的值 (对象中保存的数据，内存地址中的二进制数据)，而 is 比较对象的标识 (内存地址)。

我们最常用的是 == 比较值，然而，在变量与单列值之间比较时，应该使用 is。目前，最常使用 is 检测变量绑定值是不是 None：

```
x is None      # 判定
x is not None  # 否定写法
```

is 运算符比 == 速度快，因为它不能重载，所以 Python 不用寻找并调用特殊方法，而是直接比较整数 ID。而 a==b 是语法糖，等同于 a.__eq__(b)。继承自 object 的 __eq__ 比较两个对象的 ID，结果与 is 一样，但是多数内置类型使用更有意义的方法覆盖了该方法，会考虑对象属性的值。相等性测试可能涉及大量处理工作，例如，比较大型集合或嵌套层级深的结构时。

8.2.2 元组的相对不可变性

元组与多数 Python 集合 (列表，字典，集，等等) 一样，保存的是对象的引用。[1]如果引用的元素是可变的，即使元组本身不可变，元素依然可变。也就是说，元组的不可变性，是指元组存储的引用不可变，与引用的对象无关。

元组的相对不可变性解释了之前的谜题，这也是有些元组不可散列的原因。

8.3 默认做浅复制

复制列表 (多数内置的可变集合) 最简单的方式是使用内置的类型构造方法：

```
>>> l1 = [3,[55,44],(7,8,9)]
>>> l2 = list(l1)     # 创建副本
>>> l2
[3,[55,44],(7,8,9)]
>>> l1 == l2          # 副本与源列表相等
True
>>> l1 is l2          # 二者指代不同的对象
False
```

对列表和其它可变序列来说，还能使用更为简洁的 l2 = l1[:] 语句创建副本。

然而，构造方法或者 [:] 做的是浅复制 (即复制了最外层的容器，副本中的元素是源容

[1] str, bytes, array.array 等单一类型序列是扁平的，它们保存的不是引用而是在连续内存中数据本身

器中元素的引用)。如果所有元素都是不可变的，这样没有问题还节省内存。但如果有可变元素，这就会导致意想不到的问题。

```
# 例8-6: 浅复制
l1 = [3,[66,55,44],(7,8,9)]
l2 = l1[:]
l1.append(100)    # 在 l1 中追加元素，对 l2 没有影响
l1[1].remove(55)  # 在 l1 中删除 l1[1] 中的元素，这对 l2 有影响
print(l1)         # [3, [66, 44], (7, 8, 9), 100]
print(l2)         # [3, [66, 44], (7, 8, 9)]
l2[1] += [33,22]  # 对可变对象来说，+= 表示就地修改，这样的修改会影响到 l1
l2[2] += (10,11)  # 对元组这个扁平对象，他的修改会创新新的元组重新绑定给 l2 不会影响 l1
print(l1)         # [3, [66, 44, 33, 22], (7, 8, 9), 100]
print(l2)         # [3, [66, 44, 33, 22], (7, 8, 9, 10, 11)]
```

对任意对象做深复制和浅复制

浅复制没什么问题，但有时我们需要的是深复制 (即副本不共享内部对象的引用)。copy 模块提供的 deepcopy 和 copy 函数能为任意对象做深复制和浅复制。

```
# 例8-8: 深复制
import copy

class Bus:
    '''汽车类，可以转载乘客'''
    def __init__(self,passengers=None) -> None:
        if passengers is None:
            self.passengers = []
        else:
            self.passengers = list(passengers)

    def pick(self,name):
        self.passengers.append(name)

    def drop(self,name):
        self.passengers.remove(name)

bus1 = Bus(['Alice','Bill','Claire','David'])
bus2 = copy.copy(bus1)              # bus2 是 bus1 的浅复制
bus3 = copy.deepcopy(bus1)          # bus3 是 bus1 的深复制
print(id(bus1),id(bus2),id(bus3))   # 1391927464840 1391927489288 1391927490504
bus1.drop('Bill')
print(bus2.passengers)              # ['Alice', 'Claire', 'David']
for bus in [bus1,bus2,bus3]:        # bus1 bus2 实际上共享一个列表对象
    print(id(bus.passengers),end=' ') # 1391967343432 1391967343432 1391927554568
    print()
print(bus3.passengers)              # ['Alice', 'Bill', 'Claire', 'David']
```

一般来说，深复制不是件简单的事。如果对象有循环引用，那么这个朴素的算法会进入无限循环。deepcopy 函数会记住已经复制的对象，因此能更优雅地处理循环引用。

```
# 例8-10: deepcopy 处理循环引用
from copy import deepcopy
```

```
3
4  a = [10,20]
5  b = [a,30]
6  a.append(b)
7  print(a)   # [10, 20, [[...], 30]]
8  c = deepcopy(a)
9  print(c)   # [10, 20, [[...], 30]]
```

此外，深复制有时可能太深了。例如，对象可能会引用不该复制地外部资源或单列值。我们可以实现特殊方法 `__copy__` 和 `__deepcopy__`，控制对应的行为。

8.4 函数的参数作为引用时

Python 唯一支持的参数传递模式是共享传参，多数面向对象语言都采用这一模式。共享传参指函数的各个形式参数获得参数中各个引用的副本。也就是说，函数内部的形参是实参的别名。

这种方案的结果是，函数可能会修改作为参数传入的可变对象，但是无法修改那些对象的标识 (即不能把一个对象替换成另一个对象)。下面看一个例子：

```
1  # 例8-11: 函数传参
2  def f(a,b):
3      a += b
4      return a
5
6  x = 1
7  y = 2
8  print(f(x,y)) # 3
9  print(x,y)    # 1 2  值没有变
10
11 a = [1,2]
12 b = [3,4]
13 print(f(a,b)) # [1,2,3,4]
14 print(a,b)    # [1,2,3,4] [3,4] 列表变了
15
16 t = (1,2)
17 u = (3,4)
18 print(f(t,u)) # (1,2,3,4)
19 print(t,u)    # (1,2) (3,4)     元组没有变
```

8.4.1 不要使用可变类型作为参数的默认值

可选参数可以有默认值，这样我们的 API 在进化的同时能保证向后兼容。然而，我们应该避免使用可变的对象作为参数的默认值。

下面我们定义了一个类似例 8-8 的类，修改了 `__init__` 方法，将 `passengers` 的默认值改成 `[]` 而不会是 `None`。这样似乎就不用 `if` 判断了，但麻烦随之而生。

```python
# 例8-12: 可变默认参数的危险
class HauntedBus:
    def __init__(self,passengers=[]) -> None:
        self.passengers = passengers # 这个语句将 self.passengers 变成 passengers 的别名

    def pick(self,name):
        self.passengers.append(name)

    def drop(self,name):
        self.passengers.remove(name)

# bus1 没什么问题
bus1 = HauntedBus(['Alice','Bill'])
print(bus1.passengers)    # ['Alice', 'Bill']
bus1.pick('David')
bus1.drop('Alice')
print(bus1.passengers)    # ['Bill', 'David']

# bus2 没什么问题，因为是第一次使用默认参数的构造函数
bus2 = HauntedBus()
bus2.pick('Alice')
print(bus2.passengers)    # ['Alice']

# bus3 出问题了，它再次使用了默认参数的构造函数
bus3 = HauntedBus()
print(bus3.passengers)    # ['Alice']
```

上面这个例子的问题出在，bus2 和 bus3 实际上维护了同一张乘客表。出现这个问题的根源是：默认值在定义函数时计算 (通常在加载模块时)，因此默认值变成了函数对象的属性。也就是说，如果默认值使用可变对象，所有使用默认值构造的函数的对应属性都是同一个对象的别名。因此，如果修改了默认参数的值，那么后续的函数调用都将受影响。

```python
>>> dir(HauntedBus.__init__.__defaults__)
(['Alice'])    # 默认参数被改变
>>> HauntBus.__init__.__defaults__[0] is bus2.passengers
True           # bus2.passengers 是个别名
```

为什么 Python 会这样设计？据笔者个人猜测，这样是为了提高性能，仅在导入模块时将默认变量初始化一次，而后的函数调用只需要取对应默认变量的值即可，这样可以避免每次调用函数时都初始化一次默认参数。

8.4.2 防御可变参数

如果定义的函数接收可变参数，应该谨慎考虑调用方是否期望修改传入的参数。

让我们看一个反面例子，下车的人在队伍列表中也消失了。

```python
# 例8-15: 接收可变参数的风险
class TwilightBus:
    def __init__(self,passengers = None):
        if passengers is None:
```

```
            self.passengers = []
        else:
            self.passengers = passengers # 这个赋值语句将 self.passengers 变成 passengers 的别名

    def pick(self,name):
        self.passengers.append(name)

    def drop(self,name):
        self.passengers.remove(name)

team = ['Alice','David','Mike']
bus = TwilightBus(team)
bus.drop('Alice')
print(team)    # ['David', 'Mike']
```

上面的问题就在于，bus 中的 self.passengers 实际上与 team 共享了同一张表。正确的做法是，校车应该子集维护一张单独的乘客表，修改的方式也很简单：将参数值的副本传给 self.passengers。

```
def __init__(self,passengers = None):
    if passengers is None:
        self.passengers = []
    else:
        self.passengers = list(passengers)
```

此外还有一点要注意的是，上面代码用的是浅复制，因为列表中所有元素都是扁平的，如果不是，可能要用 copy.deepcopy() 进行深赋值。[2]

8.5 del 和垃圾回收

del 语句删除名称，而不是对象。仅当删除的变量保存的是对象的最后一个引用，或无法得到对象[3]时，del 命令会导致对象被当作垃圾回收。重新绑定也可能会导致对象的引用数量归零，导致对象被销毁。

在 CPython 中，垃圾回收使用的主要算法时引用计数。实际上，每个对象都会统计有多少个引用指向自己。当引用计数归零时，对象就立即被销毁：CPython 调用 __del__ 方法，然后释放分配给对象的内存。

为了演示对象生命结束时的情形，下面使用 weakref.finalize 注册一个回调函数，在销毁对象时调用。

```
# 例8-16: del 语句
import weakref

s1 = {1,2,3}
s2 = s1
```

[2]这段话为笔者自己加的。

[3]两个对象相互引用时，当它们的引用只存在于两者之前，垃圾回收程序会判定它们都无法获取，进而把它们都销毁。

```
6
7  def bye():        # 这个函数一定不能是要销毁的对象的绑定方法，否则会有一个指向对象的引用
8      print('Gone with the wind...')
9
10 ender = weakref.finalize(s1,bye) # 在 s1 引用上注册 bye 回调
11 print(ender.alive) # True: 在调用 finalize 对象之前，.alive 属性为 True
12
13 s2 = 'spam'       # Gone with the wind...: 此时 s2 重新绑定了一个对象，原来的对象被释放了
14 print(ender.alive) # False
```

上例明确指出 `del` 不会删除对象，但是执行 `del` 操作后可能会导致对象不可获取，从而被删除。

8.6 弱引用

正是因为有引用，对象才会在内存中存在。当对象的引用数量归零后，垃圾回收程序会把对象销毁。但是，有时需要引用对象，而不让对象存在的时候超过所需的时间。这经常用在缓存中。

弱引用是可调用的对象，返回的是被引用的对象；如果所指对象不存在了，返回 `None`。

弱引用不会增加对象的引用数量。引用的目标对象称为所指对象。因此我们说，弱引用不会妨碍所指对象被当作垃圾回收。

弱引用在缓存应用中很有用，因为我们不想仅因为被缓存引用着而始终保存缓存对象。

下面示例展示了如何使用 `weakref.ref` 实例获取所指对象。如果对象存在，调用弱引用可以获取对象，否则返回 `None`。

```
1  # 例8-17: 弱引用
2  import weakref
3
4  a_set = {0,1}
5  wref = weakref.ref(a_set) # 创建弱引用对象 wref，下一行审查它
6  print(wref)       # <weakref at 0x0000014415BF4E58; to 'set' at 0x0000014415C0A668>
7  print(wref())     # {0,1}: 调用 wref() 返回的是被引用的对象
8  a_set = {2,3,4}   # 将 a_set 绑定到新的对象
9  print(wref() is None) # True: 对象不存在，弱引用返回 None
```

`weakref` 模块的文档指出，`weakref.ref` 类其实是低层接口，供高级用途使用，多数程序最好使用 `weakref` 集合和 `finalize`。

8.6.1 `WeakValueDictionary` 简介

`WeakValueDictionary` 类实现的是一种可变映射，里面的值是对象的弱引用。被引用的对象在程序中的其他地方被当作垃圾回收后，对应的键会自动从 `WeakValueDictionary` 中删除。因此 `WeakValueDictionary` 经常用于缓存。

```
1  # 例8-18: WeakValueDictionary
```

```
import weakref

class Cheese:
    def __init__(self,kind) -> None:
        self.kind = kind

    def __repr__(self) -> str:
        return 'Cheese(%r)' % self.kind

stock = weakref.WeakValueDictionary()
catalog = [Cheese('Red Leicester'), Cheese('Tilsit'),Cheese('Brie'),Cheese('Parmesan')]

for cheese in catalog:    # 将奶酪的名称映射到 catalog 中 Cheese 实例的弱引用上
    stock[cheese.kind] = cheese

print(sorted(stock.keys())) # ['Brie', 'Parmesan', 'Red Leicester', 'Tilsit']
del catalog
print(sorted(stock.keys())) # ['Parmesan']:删除 catalog 后，大多数奶酪不见了，但不是所有
del cheese
print(sorted(stock.keys())) # []:因为 for 循环中的 cheese 也是全局变量，删掉就没事了
```

`weakref` 模块还有很多有用的方法，请读者自行查阅文档了解。

8.6.2 弱引用的局限

不是每个 Python 对象都可以作为弱引用的目标。基本的 `list` 和 `dict` 实例不能作为所指对象，但是它们的子类可以轻松地解决这个问题。

```
class MyList(list):
    '''list 的子类，实例可以作为弱引用的目标'''

a_list = MyList(range(10))
wref_to_a_list = weakref.ref(a_list)
```

`set` 实例可以作为所指对象。但是 `int` 和 `tuple` 实例不能作为弱引用的目标，子类也不行。

8.7 Python 对不可变类型施加的把戏

本节讨论的是 Python 的实现细节，可以跳过。

对于元组 `t` 来说，`t[:]` 不创建副本，而是返回同一个对象的引用。此外，`tuple(t)` 获得的也是同一个元组的引用。

`str`，`bytes`，`frozenset` 也有这种行为。其中 `frozenset` 实例不是序列，因此不能使用 `[:]`，但是 `fs.copy()` 具有相似的效果。

共享字符串字面量是一种优化措施，称为驻留。CPython 还会在小的整数上使用这个优化措施，防止重复创建“热门”数字。但并不是所有字符串和数字都会采用驻留，这是实现细

节，没有文档说明。

这些令人难以记作的操作是 Python 为了节省内存，提升速度的优化行为。这并不会带来任何麻烦，因此只有不可变类型会受到影响，我们平时也无需在意这些细枝末节[4]。

[4]装逼的时候可以

9 符合 Python 风格的对象

9.1 对象表示形式

每门面向对象的语言至少含有一种获取对象的字符串表示形式和标准方法。Python 提供了两种方法。

- `repr()`
  便于开发者理解的方法，返回对象的字符串表示形式。
- `str()`
  便于用户理解的方法，返回对象的字符串表示形式。默认 `print()` 打印对象时调用，如果 `__str__` 没有实现，则调用 `__repr__` 代替。

为了给对象提供其他的表示形式,还会用到另外两个特殊方法:`__bytes__` 和 `__format__`。`__bytes__` 与 `__str__` 类似:`bytes()` 函数调用它获取对象的字符序列表示形式。而 __format__ 方法会被内置的 `format()` 函数和 `str.format()` 方法调用，使用特殊的格式代码显示对象的字符串表示形式。

9.2 再谈向量类

下面我们构建一个 `Vector2d` 类来说明对象表示形式，说明均在代码中给出。

```python
# 例9-1: Vector2d 的多种表示形式
from array import array
import math

class Vector2d:
    typecode = 'd'

    def __init__(self,x,y) -> None:
        self.x = float(x)
        self.y = float(y)

    '''实现迭代，从而实现拆包'''
    def __iter__ (self):
        return (i for i in (self.x,self.y))

    '''{!r} 获取各个分量的表示形式'''
    def __repr__(self) -> str:
        class_name = type(self).__name__ # type(self).__name__ 是类名，self.__name__  对象名
        return '{}({!r},{!r})'.format(class_name,*self)

    def __str__(self) -> str:
        return str(tuple(self))

    def __bytes__(self):
        return (bytes([ord(self.typecode)]) + bytes(array(self.typecode,self)))
```

```
    '''有个缺陷：其他可迭代对象也可进行比较'''
    def __eq__(self, other) -> bool:
        return tuple(self) == tuple(other)

    def __abs__(self):
        return math.hypot(self.x , self.y)

    def __bool__(self):
        return bool(abs(self))

v1 = Vector2d(3,4)
print(v1)                # (3.0, 4.0): 调用了 __str__
x,y = v1
print(x,y)               # 3.4 4.0: 调用了 __iter__ 进行拆包
print(repr(v1))          # Vector2d(3.0,4.0)
v1_clone = eval(repr(v1)) # repr 函数调用 Vector2d 实例得到的是对构造方法的准确表述
print(v1 == v1_clone)    # True
print(bytes(v1))         # b'd\x00\x00\x00\x00\x00\x00\x08@\x00\x00\x00\x00\x00\x00\x10@'
print(abs(v1))           # 5.0
print(bool(v1))          # True
```

9.3 备选构造方法

我们可以把 Vector2d 实例转换成字节序列，同理，也应能从字节序列转换成 Vector2d 实例。在标准库中 array.array 有个类方法 .frombytes 符合要求。下面为 Vector2d 定义一个同名类方法。

```
# 例9-3: 定义备选构造方法
@classmethod  # 类方法装饰器，下一节将说明
def frombytes(cls,octets): # 不传入 self 参数，相反通过 cls 传入类本身
    typecode = char(octets[0]) # 从第一个字节中读取 typecode
    memv = memoryview(octets[1:]).cast(typecode) # 使用传入的 octets 字节序列创建一个
        memoryview，然后使用 typecode 转换
    return cls(*memv)     # 拆包转换后的 memoryview，得到构造方法所需的一对参数
```

9.4 classmethod 和 staticmethod

classmethod 用于定义操作类，而不是操作实例的方法。classmethod 改变了调用方法的方法，因此类方法的第一个参数是类本身 (cls)，而不是实例 (self)。classmethod 最常见的用途是定义备选构造方法。例如上一个例子 frombytes。注意，frombytes 的最后一行使用 cls 参数构建了一个新实例，即 cls(*memv)。按照约定，类方法的第一个参数名为 cls。

staticmethod 装饰器也会改变方法的调用方法，但是第一个参数不是特殊的值。其实，静态方法就是普通的方法，只是碰巧定义在类中，而不是在模块里。

```python
# 例9-4: 比较 classmethod 和 staticmethod
class Demo:
    @classmethod
    def klassmeth(*args):
        return args

    @staticmethod
    def statmeth(*args):
        return args

    def fun(*args):
        return args

# 不管怎样调用 klassmeth，它的第一个参数始终是 Demo 类
print(Demo.klassmeth())          # (<class '__main__.Demo'>,)
print(Demo.klassmeth('spam'))    # (<class '__main__.Demo'>, 'spam')
# statmeth 的行为与普通的函数相似
print(Demo.statmeth())           # ()
print(Demo.statmeth('spam'))     # ('spam',)
# 普通函数
print(Demo.fun())                # ()
print(Demo.fun('spam'))          # ('spam',)
```

`classmethod` 装饰器的作用毋庸置疑，而 `staticmethod` 装饰器的作用却显得相形见绌，部分人认为 `staticmethod` 并不是十分必要[5]。

9.5 格式化显示

内置的 `format()` 函数和 `str.format()` 方法把各个类型的格式化方法委托给相应的 `.__format__` 方法。`format_spec` 是格式说明符，它是：

- `format(my_obj,format_spec)` 的第二个参数，或者
- `str.format()` 方法的格式字符串，`{}` 里代换字段中冒号后面的部分

```
>>> brl = 1/2.43
>>> brl
0.4115226337448559
>>> format(brl,'0.4f')    # 说明符是 0.4f
'0.4115'
>>> '1BRL = {rate:0.2f} USD'.format(rate=brl) # 说明符是 0.2f，代换字段中的 'rate'
    子串是字段名称
'1BRL = 0.41 USD'
```

上例第 6 行指出了一个重要的知识点，'0.mass:5.3e' 这样的格式字符串其实包括两部分，冒号左边的'0.mass' 在代换字段句法中式字段名，冒号后面的'5.3e' 式格式说明符。格式说明符使用的表示法叫做格式规范微语言。

格式规范微语言为一些内置类型提供了专门的表示代码。比如，`b` 和 `x` 分别表示二进制

[5]原作者这样认为

和十六进制的 int 类型，f 表示小数形式的 float 类型。而% 表示百分数形式：

```
>>> format(42,'b')
'101010'
>>> format(2/3,'.1%')
'66.7%'
```

格式规范微语言式可扩展的，因为各个类可以自行决定如何解释 format_spec 参数。例如，datetime 模块中的类，它们的 __format__ 方法使用的格式代码与 strftime() 函数一样。下面是几个例子。

```
>>> from datetime import datetime
>>> now = datetime.now()
>>> format(now,'%H:%M:%S')
'18:49:05'
>>> "It is now {:%I:%M %p}".format(now)
"It's now 06:49 PM"
```

如果类没有定义 __format__ 方法，从 object 继承的方法会返回 str(my_object)。我们为 Vector2d 类定义了 __str__ 方法，因此可以这样做：

```
>>> v1 = Vector2d(3,4)
>>> format(v1)
'(3.0,4.0)'
```

然而，如果传入格式说明符，object.__format__ 方法会抛出 TypeError:

```
>>> format(v1,'.3f')
TypeError:......
```

我们将实现自己的微语言来解决这个问题。首先，假设用户提供的格式说明符式用于格式化向量中的各个浮点数分量的。我们想达到的效果是：

```
>>> v1 = Vector2d(3,4)
>>> format(v1)
'(3.0, 4.0)'
>>> format(v1,'.2f')
'(3.00, 4.00)'
>>> format(v1,'.3e')
'(3.000e+00, 4.000e+00)'
```

实现这种输出的 __format__ 方法如下例所示：

```
def __format__(self, fmt_spec=''):
    components = (format(c,fmt_spec) for c in self)
    return '({},{})'.format(*component)
```

下面要在微语言中添加一个自定义的格式代码：如果格式说明符以’p’ 结尾，那么在极坐标中显示向量，即 <r,θ>。

下面定义一个计算 angle 的方法。

```
def angle(self):
    return math.atan2(self.y,self.x)
```

这样便可以增强 __format__ 方法，计算极坐标。

```
def __format__(self, fmt_spec='') -> str:
if fmt_spec.endswith('p'):   # 极坐标系坐标
    fmt_spec = fmt_spec[:-1] # 删除 p 后缀
    coords = (abs(self),self.angle())
    outer_fmt = '<{},{}>'
else:                        # 平面直角坐标系坐标
    coords = self
    outer_fmt = '({},{})'
components = (format(c,fmt_spec) for c in coords) #
    使用各个分量生成可迭代的对象，构成格式化字符串
return outer_fmt.format(*components) # 把格式化字符串带入外层格式
```

9.6 可散列的 Vector2d

按照目前的定义，Vector2d 是不可散列的，应为没有实现 __hash__ 方法 (__eq__ 方法已实现)。此外，还要让向量不可变。

为了让向量不可变，需要将变量变成私有的，其方法是在变量前加两个 _。

```
# 例9-7: 私有的 Vector2d
class Vector2d:
    typecode = 'd'

    def __init__(self,x,y) -> None:
        self.__x = float(x)
        self.__y = float(y)

    @property      # 装饰器: 读值方法
    def x(self):   # 读值方法与公开属性同名
        return self.__x

    @property
    def y(self):
        return self.__y

    def __iter__(self):   # 读取公开特性而不是私有属性
        return (i for i in (self.x,self.y))

    def __hash__(self) -> int:
        return hash(self.x) ^ hash(self.y)
```

让这些向量不可变才能实现 __hash__ 方法。这个方法应该返回一个整数，理想情况下还要考虑对象属性的散列值 (__eq__ 方法也要使用)，因为相等的对象应该具有相等的散列值。

根据特殊方法

texttt__hash__ 的文档，最好使用位运算符异或混合各分量的散列值。

```
def __hash__(self) -> int:
    return hash(self.x) ^ hash(self.y)
```

添加了 `__hash__` 方法之后，向量变成可散列的了。

```
>>> v1 = Vector2d(3,4)
>>> v2 = Vector2d(3.1,4.2)
>>> hash(v1)
7
>>> hash(v2)
38430......
```

要创建可散列的类型，不一定要实现特性，也不一定要保护实例属性。只需正确地实现 `__hash__` 和 `__eq__` 方法。但是，实例的散列值绝不应该变化，因此我们借机提到了只读属性。

9.7 私有属性和受保护属性

Python 没有像 Java 那样使用 `private` 修饰符创建的私有属性，但它有个简单的机制能够避免子类意外覆盖 “私有” 属性。

首先，我们要将属性设置为私有属性的形式，即前面加两个下划线，尾部没有或至多一个下划线。Python 会将此类属性名存入实例的 `__dict__` 属性中，而且会在前面加上一个下划线和类名，用于区分父类和子类的属性。这个语言特性叫名称改写。

以之前定义的 `Vector2d` 为例：

```
>>> v1 = Vector2d(3,4)
>>> v1.__dict__
{'_Vector2d__y':4.0, '_Vector2d__x':3.0}
>>> v1._Vector2d__x
3.0
```

名称改写是一种安全措施，不能保证万无一失：他的目的是避免意外访问，不能防止故意做错事。

例如上一个例子中，只要编写 `v1._Vector2d__x = 7` 这样的代码，就能轻松地位 `Vector2d` 实例地私有分量直接赋值。

不是所有人都喜欢名称改写功能。也不是所有人都喜欢 `self.__x` 这种名称。对于不喜欢这种句法的人，他们约定使用一个下划线前缀编写 “受保护” 的属性。而其他人认为应该使用命名约定来避免意外覆盖属性。

Python 解释器不会到使用单个下划线的属性名做特殊处理，不过这是很多 Python 程序员的约定，他们不会在类外部访问这种属性。Python 文档的一些角落将这种写法的属性称为 “受保护” 的属性，不过并不是所有人都这样叫。

9.8 使用 __slots__ 类属性节省空间

默认情况下，Python 在各个实例中名为 __dict__ 的字典里存储实例属性。这会大大提高访问速度，但字典会消耗大量内存。如果要处理属性不多的大量实例，则可通过 __slots__ 类属性，能节省大量内存空间，方法是让解释器在元组中存储实例属性，而不是字典。

定义 __slots__ 的方式是：创建一个类属性，使用 __slots__ 这个名字，并把它的值设为一个字符串构成的可迭代对象，其中各个元素表示各个实例属性。原作者倾向于使用元组，因为这样定义的 __slots__ 中含有的信息不会变化。

```
class Vector2d:
    __slots__ = ('__x','__y')

    # 各个方法的实现省略
```

在类中定义 __slots__ 属性的目的是告诉解释器：这个类中所有的实例属性都在这了。这样 Python 会用类似元组的结构存储实例变量。此外，在类中定义 __slots__ 属性后，实例不能再有 __slots__ 中所列名称之外的其他属性。

在作者的实验中，创建了 10,000,000 个 Vector2d 实例，使用 __dict__ 属性时，RAM 用量最高为 1.5GB。而使用 __slots__ 则为 655 MB。

还有一种神奇的用法，如果将 __dict__ 这个名称加到 __slots__ 中，实例会在元组中保存各个实例的属性，此外还支持动态创建属性，这些属性存储在常规的 __dict__。但是这违背了 __slots__ 设计的初衷 (节省内存)。

__slots__ 的问题

如果使用得当，__slots__ 能显著节省内存，不过有几点要注意。

- 每个子类都要定义 __slots__ 属性，因为解释器会忽略继承的 __slots__ 属性。
- 实例只能拥有 __slots__ 中列出的属性。
- 如果不把 '__weakref__' 加入 __slots__ ，实例就不能作为弱引用的目标。

9.9 覆盖类属性

Python 有个独特的特性:类属性可用于为实例属性提供默认值。Vector2d 中有个 typecode 类属性，__bytes__ 方法两次用到了它，而且故意使用 self.typecode 读取它的值。因为 Vector2d 实例本身没有 typecode 属性,所以 self.typecode 默认获取的是 Vector2d.typecode 类属性的值。

但是，如果为不存在的实例属性赋值，会新建实例属性。假如我们为 typecode 实例属性赋值，那么同名类属性不受影响。然而，自此以后，实例读取的 self.typecode 是实例属性 typecode，也就是把同名类属性遮盖了。借助这一特性，可以为各个实例的 typecode 属性定制不同的值。

如果想修改类属性的值，必须直接在类上修改，不能通过实例修改。如果要修改所有实

例 (没有 typecode 实例变量) 的 typecode 属性的默认值，可以这样做：

```
>>> Vector2d.typecode = 'f'
```

然而，有种修改方法更符合 Python 风格，而且效果持久，也更具针对性。类属性是公开的，因此会被子类继承，以实经常会创建一个子类，只用于定制类的数据属性。具体做法如下：

```
class ShortVector2d(Vector2d):
    typecode = 'f'
```

这也说明了在 Vector2d.__repr__ 方法中为什么没有硬编码 class_name 的值，而是使用 type(self).__name__ 获取。

```
# 在 Vector2d 类中定义
def __repr__(self):
    class_name = type(self).__name__
    return '{}({!r},{!r})'.format(class_name. *self)
```

如果硬编码 class_name 的值，那么 Vector2d 的子类要覆盖 __repr__ 方法，只是为了修改 class_name 的值。从实例的类型中读取类名，__repr__ 方法就可以放心继承。

10 序列的修改，散列和切片

不要检查它是不是鸭子：检查它的叫声像不像鸭子，它的走路姿势像不像鸭子，等等。具体检查什么取决于你想使用语言的哪些行为。

——Alex Martelli

这一章我们会基于前面的 `Vector2d`，向前迈出一大步，定义表示多为向量的 `Vector` 类。这个类的行为与 Python 中标准的不可变扁平序列一样。

10.1 Vector 类：用户定义的序列类型

我们将使用组合模式实现 `Vector` 类，而不使用继承。向量的分量存储在浮点数数组中，而且还将实现不可变扁平序列所需的方法。

不过，在实现序列方法之前，我们要确保 `Vector` 类与前一章定义的 `Vector2d` 类兼容，除非有些地方让二者兼容没有什么意义。

10.2 Vector 类第 1 版：与 Vector2d 兼容

我们会故意不让 `Vector` 的构造方法与 `Vector2d` 的构造方法兼容。我们让 `__init__` 方法接受任意个参数 (通过 `*args`)。但是序列的构造方法最好接受可迭代的对象作为参数，因为所有内置的序列类型都是这样做的。下面是我们需要达到的效果：

```
>>> Vector([3.1,4.2])
Vector([3.1,4.2])
>>> Vector((3,4,5))
Vector([3.0,4.0,5.0])
>>> Vector(range(10))
Vector([0.0,1.0,2.0,3.0,...])
```

除了新构造方法的签名外，还确保了传入两个分量时，`Vector2d` 类的每个测试都能通过，而且得到相同的结果。

```
from array import array
import reprlib
import math

class Vector:
    typecode = 'd'

    def __init__(self,components):
        self._components = array(self.typecode,components) # 受保护的属性

    def __iter__(self):
        return iter(self._components)    # iter 会在下面的章节讲解

```

```
    def __repr__(self):
        components = reprlib.repr(self._components) # reprlib.repr() 获取有限长度表示形式
        components = components[components.find('['):-1] # 去掉前面的 array('d')[ 和后面的 ]
        return 'Vector({})'.format(components)

    def __str__(self):
        return str(tuple(self))

    def __bytes(self):
        return (bytes[ord(self.typecode)]) + bytes(self._components)

    def __eq__(self,other):
        return tuple(self) == tuple(other)

    def __abs__(self):
        return math.sqrt(sum(x*x for x in self))

    def __bool__(self):
        return bool(abs(self))

    @classmethod
    def frombytes(cls,octets):
        typecode = chr(octets[0])
        memv = memoryview(octets[1:]).cast(typecode)
        return cls(memv)
```

其中 `reprlib.repr` 的时候需要做些说明。这个函数用于生成大型结构或递归结构的安全表达式，它会限制输出字符串的长度，用'...' 表示截断的部分。我们实例的表示形式是 `Vector([3.0,4.0,5.0])` 这样的，而不是 `Vector(array('d',[3.0,4.0,5.0]))`，因为 `Vector` 实例中的数组是实现细节，没必要显示。

编写 `__repr__` 方法时，本可以使用 `reprlib.repr(list(self._components))` 。然而，这么做有点浪费，因为要把 `self._components` 中的每个元素复制到一个列表中，然后使用列表的表示形式。

10.3 协议和鸭子类型

在第一章中说过，在 Python 中创建功能完善的序列类型无需使用继承，只需实现符合序列协议的方法。

在面向对象编程中，协议是非正式的接口，只在文档中定义，在代码中不定义。例如，Python 的序列协议只需要 `__len__` 和 `__getitem__` 两个方法。任何类只要使用标准的签名和语义实现了这两个方法，就能用在任何期待序列的地方。以示例 1-1 为例：

```
# 例1-1: 几个内置函数的理解
import collections

Card = collections.namedtuple('Card', ['rank', 'suit'])

```

```
class FrenchDeck:
    ranks = [str(n) for n in range(2, 11)] + list('JQKA')
    suits = 'spades diamonds clubs hearts'.split()

    def __init__(self):
        self._cards = [Card(rank, suit)
                       for suit in self.suits for rank in self.ranks]

    def __len__(self): # 内置函数 len()
        return len(self._cards)

    def __getitem__(self, position): # 内置迭代器
        return self._cards[position]
```

上面例子中的 FrenchDeck 类能充分利用 Python 的许多功能，因为它实现了序列协议，不过代码中并没有声明这一点。任何有经验的程序员只要看一眼就知道它是序列。我们说它是序列，是因为它的行为像序列，这才是重点。

协议是非正式的，没有强制力的，因此如果你知道类的具体使用场景，通常只需要实现一个协议的部分。例如，为了支持迭代，只需实现
texttt__getitem__ 方法。

10.4 Vector 类第 2 版：可切片的序列

如 FrenchDeck 类所示，如果能委托给对象中的序列属性 (如 self._components 数组)，支持序列协议特别简单。下述只有一行代码的 __len__ 和 __getitem__ 方法是个好的开始。

```
class Vector:
    # 省略很多行

    def __len___(self):
        return len(self._components)

    def __getitem__(self,index):
        return self._components[index]
```

添加这两个方法后，就能执行下列操作了：

```
>>> v1 = Vector([3,4,5])
>>> len(v1)
3
>>> v1[0], v1[-1]
(3.0, 5.0)
>>> v7 = Vector(range(7))
>>> v7[1:4]
>>> array('d',[1.0,2.0,3.0])
```

现在可以实现切片功能了，但是这并不完美，我们希望 Vector 实例的切片也是 Vector 实例，而不是数组。对 Vector 来说，如果切片生成普通的数组，将会缺失大量功能。

为了实现这一目标，我们不能简单地委托给数组切片。我们要分析传给 __getitem__ 方法的参数，做适当的处理。

10.4.1 切片原理

查看下面例子：

```
>>> class MySeq:
...     def __getitem__(self, index):
...         return index  # 直接返回传给它的值
...
>>> s = MySeq()
>>> s[1]
1
>>> s[1:4]
slice(1, 4, None)
>>> s[1:4:2]
slice(1,4,2)
>>> s[1:4:2, 9]          # 如果 [] 中含有逗号，那么 __getitem__ 收到的是元组。
(slice(1,4,2), 9)
>>> s[1:4:2, 7:9]        # 元组中甚至可以有多个切片
(slice(1,4,2), slice(7,9,None))
```

我们继续深入研究 slice 本身。

```
>>> slice
<class 'slice'>
>>> dir(slice)
['__class__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__',
    '__getattribute__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le__',
    '__lt__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
    '__sizeof__', '__str__', '__subclasshook__', 'indices', 'start', 'step', 'stop']
```

由此，我们发现 slice 是内置的类型，他有 start，step，stop 三个属性，以及 indices 方法。这个方法有很大作用但鲜为人知。

```
>>> help(slice.indices)
S.indices(len) -> (start, stop, stride)
```

给定长度为 len 的序列，计算 S 表示的扩展切片的起始 (start) 和结尾 (stop) 索引，以及步幅 (stride)。超出边界的索引会被截掉。

换句话说，indices 方法开放了内置序列实现的棘手逻辑，用于优雅地处理确实索引和负数索引，以及长度超过目标序列的切片。这个方法会“整顿”元组，把 start，stop 和 stride 都变成非负数，并且都落在指定长度序列的边界内。

下面举几个例子，假设有个长度为 5 的序列，例如’ABCDE’:

```
>>> slice(None,10,2).indices(5)  # 'ABCDE'[:10:2] 等同于 'ABCDE'[0:5:2]
(0,5,2)
>>> slice(-3,None,None).indices(5) # 'ABCDE'[-3:] 等同于 'ABCDE'[2:5:1]
```

```
(2,5,1)
```

在 Vector 类中无需使用 slice.indices() 方法，因为收到切片参数时，我们会委托 _components 数组处理。但是，如果没有底层序列类型作为依靠，那么使用这个方法能节省大量时间。

10.4.2 能处理切片的 __getitem__ 方法

下面给出让 Vector 表现为序列所需的两个方法：__len__ 和 __getitem__：

```
def __len__(self):
    return len(self._components)
def __getitem__(self,index):
    cls = type(self)  # 获取实例所属类(即 Vector)
    if isinstance(index,slice):  # 如果 index 参数是 slice 对象
        return cls(self._components[index]) # 调用类的构造方法，使用 _components
            数组的切片构建一个新 Vector 实例
    elif isinstance(index,numbers.Integral): # 如果 index 是 int 或其他整数类型，返回对应元素
        return self._components[index]
    else:                                    # 其他情况抛出异常
        msg = '{cls.__name__} indices must be integers'
        raise TypeError(msg.format(cls=cls))
```

10.5 Vector 类第 3 版：动态存取属性

Vector2d 变成 Vector 之后，就没办法通过名称访问向量的分量了 (如 v.x，v.y)。现在我们处理的向量可能有大量分量。不过，弱能通过单个字母访问前几个分量的话比较方便。比如，用 x,y,z 代替 v[0],v[1],v[2]。

当我们查找某一属性查找失败后,解释器会调用 __getattr__ 方法。简单来说,对 my_obj.x 表达式，Python 会检查有没有名为 x 的属性；如果没有，到类中 (my_obj.__class__) 查找；如果还没有，顺着继承树继续查找。[6]如果依旧找不到，调用 my_obj 所属类中定义的 __getattr__ 方法，传入 self 和属性名称的字符串形式 (如 'x')。

下面是为 Vector 定义的 __getattr__ 方法。这个方法的作用很简单，它检查所查找的属性是不是 xyzt 中的某个字母，如果是，则返回对应分量。

```
shortcut_names = 'xyzt'

def __getattr__(self,name):
    cls = type(self)
    if len(name) == 1:
        pos = cls.shortcut_names.find(name)
        if 0 <= pos < len(shortcut_names):
            return self._components[pos]
    msg = '{.__name__!r} object has no attributes {!r}'
```

[6]属性查找机制非常复杂，会在后面章节说明。

```
    raise AttributeError(msg.format(cls,name))
```

__getattr__ 方法的实现并不难，但仅这样实现还不够：

```
>>> v = Vector(range(5))
>>> v
Vector([0.0, 1.0, 2.0, 3.0, 4.0])
>>> v.x
0.0
>>> v.x = 10
>>> v.x
10
>>> v
Vector([0.0, 1.0, 2.0, 3.0, 4.0]) # 多西得？
```

上面例子之所以矛盾，是 __getattr__ 的运作方式导致的：仅当对象没有指定名称的属性时，Python 才会调用那个方法，这是一种后备机制。像 v.x = 10 这样赋值之后，v 对象有了 x 属性。因此使用 v.x 获取 x 属性的值时不会调用 __getattr__ 方法了。解释器直接返回绑定到 v.x 上的值，即 10、另一方面，__getattr__ 方法的实现没有考虑到 self._components 之外的示例属性，而是从这个属性中获取 shortcut_names 中所列的“虚拟属性”。

为了避免这种前后矛盾的现象，我们要改写 Vector 类中设置属性的逻辑。

回想上一章最后一个 Vector2d 示例中，如果为 .x 或 .y 实例属性赋值，会抛出 AttributeError。为了避免歧义，在 Vector 类中，如果为名称是单个小写字母的属性赋值，我们也想抛出那个异常。为此，我们要实现 __setattr__ 方法。

```
def __setattr__(self, name, value):
    cls = type(self)
    if len(name) == 1: # 当处理的是单个字符的属性
        if name in cls.shortcut_names:
            error = 'readonly attribute {attr_name!r}'
        elif name.islower():  # 当是其他小写字母时
            error = "can't set attributes 'a' to 'z' in {cls_name!r}"
        else:          # 否则设置错误为空串
            error = ''
        if error:      # 如果有错误，抛出 AttributeError
            msg = error.format(cls_name = cls.__name__, attr_name = name)
            raise AttributeError(msg)
    super().__setattr__(name,value) # 默认情况，在超类上调用 __setattr__ 方法
```

虽然这个示例不支持为 Vector 分量赋值，但是有一个问题要特别注意：多数时候，如果实现了 __getattr__ 方法，那么也要定义 __setattr__ 方法，以防对象的行为不一致。

如果像允许修改分量，可以使用 __setitem__ 方法，支持 v[0] = 1.1 这样的赋值，以及实现 __setattr__ 方法，支持 v.x = 1.1 这样的赋值，不过，我们要保持 Vector 是不可变的，因为在下一节，我们将把它变成可散列的。

10.6 Vector 类第 4 版：散列可快速等值测试

我们要再次实现 __hash__ 方法。加上现有的 __eq__ 方法，这会把 Vector 示例变成可散列的对象。

我们要使用异或运算符依次计算各个分量的散列值。这正是 functools.reduce 函数的作用[7]。下面示例展示了计算聚合异或的三种方法：

```
>>> n = 0
>>> for i in range(1,6):  # for 循环计算
...     n ^= i
...
>>> n
1
>>> import functools      # 使用 reduce 集合 lambda 匿名函数
>>> functools.reduce(lambda a,b: a^b, range(6))
1
>>> import operator       # 替换 lambda 函数
>>> functools.reduce(operator.xor, range(6))
1
```

这里原作者使用了第三种方法编写 Vector.__hash__ 方法。

```
import functools
import operator

......
    def __hash__(self):
        hashes = (hash(x) for x in self._components)
        return functools.reduce(operator.xor, hashes, 0)
```

使用 reduce(function, iterable, initializer) 函数时最好提供第三个参数。这样能避免这个异常：TypeError: reduce() of empty sequence with no initial value。如果序列为孔，initializer 是返回的结果；否则，在归约中使用它作为第一个参数，因此应该使用对应函数的恒等值。

上述实现的 __hash__ 方法是一种映射归约计算：把函数应用到各个元素上，生成一个新序列，然后计算聚合值。

映射过程计算各个分量的散列值，规约过程则使用 xor 运算符聚合所有散列值。把生成器表达式替换成 map 方法，映射过程更明显。

```
def __hash__(self):
    hashed = map(hash, self._components)
    return functools.reduce(operator.xor, hashed, 0)
```

既然讲到了归约，那就把前面曹操实现的 __eq__ 方法修改一下，减少处理时间和内存用量 (对大型向量来说)。之前的 __eq__ 方法非常简洁：

```
def __eq__(self, other):
```

[7]reduce 函数的作用参考 CSDN:https://blog.csdn.net/u012193416/article/details/89397388

```
    return tuple(self) == tuple(other)
```

Vector2d 和 Vector 都可以这样做，让甚至会认为 Vector([1,2]) 与 (1,2) 相等。这是个问题，但我们暂且忽略。[8]这样做对有几千个分量的 Vector 实例来说，效率十分低下。上述实现方法要完整赋值两个操作数，构建两个元组，而这么做只是为了使用 tuple 的 __eq__ 方法。对 Vector2d 来说，这是个捷径。但对 Vector 这降低了效率。下面提供一个更好的方法：

```
def __eq__(self, other):
    if len(self) != len(other):
        return False
    for a,b in zip(self,other):
        if a!=b:
            return False
    return True
```

上述效率很好，不过用于计算聚合值的整个 for 循环可以替换成一行 all 函数调用：如果所有分量对的比较结果都是 True，那么结果就是 True。只要有一次比较结果是 False，all 函数返回 False。

```
def __eq__(self, other):
    return len(self) == len(other) and all(a == b for a,b in zip(self, other))
```

10.7 Vector 类第 5 版：格式化

Vector 类的 __format__ 方法与 Vector2d 类的相似，但是不适用极坐标，而使用球面坐标。我们把自定义的格式后缀由'p' 改为'h'。

```
def angle(self, n):
    r = math.sqrt(sum(x*x for x in self[n:]))
    a = math.atan2(r, self[n-1])
    if (n == len(self)-1) and (self[-1]<0):
        return math.pi * 2 -a
    else
        return a

def angles(self):
    return (self.angle(n) for n in range(1,len(self)))

def __format__(self, fmt_spec='')
    if fmt_spec.endswith('h'):
        fmt_spec = fmt_spec[:-1]
        coords = itertools.chain([abs(self)], self.angles())
        outer_fmt = '<{}>'
    else:
        coords = self
```

[8]后面章节会解决。

```
        outer_fmt = '(<>)'
    components = (format(c, fmt_spec) for c in coords)
    return outer_fmt.format(', '.join(components))
```

11 接口：从协议到抽象基类

11.1 Python 文化中的接口和协议

在引入抽象基类之前，Python 就已经非常成功了，即使是现在也很少有代码使用抽象基类。

Python 没有 `interface` 关键字，而且除了抽象基类，每个类都有接口：类实现或继承的公开属性 (方法或数据属性)，包括特殊方法，如 `__getitem__` 或 `__add__` 。

按照定义，受保护的属性和私有属性不在接口中：即便 “受保护的” 属性只是采取命名约定实现的；私有属性可以轻松地访问，原因也是如此。不要违背这些约定。

另一方面，不要觉得把公开数据属性放入对象的接口中不妥，因为如果需要，总能实现读值方法和设值方法，把数据属性变成特性，使用 `obj.attr` 句法的客户代码不会受到影响。`Vector2d` 类就是这么做的。

```
class Vector2d:
    typecode = 'd'

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

    def __iter__(self):
        return (i for i in (self.x, self.y))
```

在上述示例中，我们把 `x` 和 `y` 变成了只读属性。这是一项重大重构，但是 `Vector2d` 的接口基本没变：用户仍能读取 `my_vector.x` 和 `my_vector.y`。

关于接口还有个实用的补充定义：对象公开方法的子集，让对象在系统中扮演特定的角色。Python 文档中的 “文件类对象” 或 “可迭代对象” 就是这个意思，这种说法指的不是特定的类。接口是实现特定角色的方法的集合，这样理解正是 Smalltalk 程序员所说的协议，其他动态语言社区都借鉴了这个术语。协议与继承没有关系。一个类可以实现多个协议，从而让实例扮演多个角色。

协议是接口，但不是正式的 (只由文档和约定定义)，因此协议不能像正式接口那样施加限制。一个类可能只实现部分接口，这是允许的。有时，某些 API 只要求 “文件类对象” 返回字节序列的 `.read()` 方法。在特定的上下文中可能需要其他文件操作方法，也可能不需要。

对 Python 程序员来说，“X 类对象” “X 协议” 和 “X 接口” 是一个意思。

11.2 Python 喜欢序列

序列协议是 Python 最基础的协议之一。即便对象只实现了那个协议最基本的一部分，解释器也会负责任地处理。

下图展示了定义为抽象基类的 `Sequence` 正式接口。

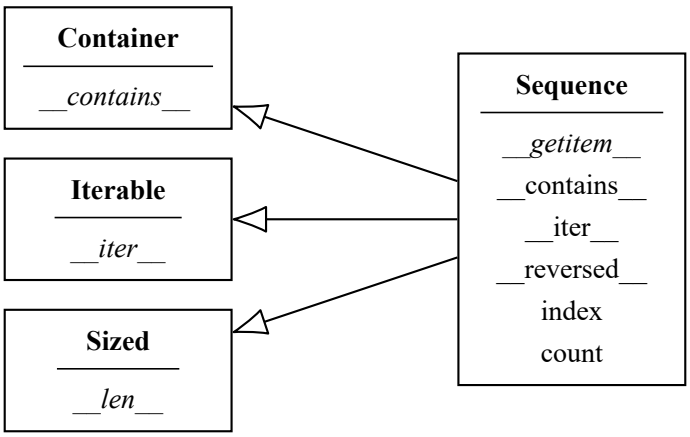

图 11.1　Sequence 抽象基类

现在看一个示例 Foo 类，它没有继承 abc.Sequence，而且只实现了序列协议的一个方法 __getitem__ (没有实现 __len__ 方法)。

```
>>> class Foo:
...     def __getitem__(self, pos):
...         return range(0, 30, 10)[pos]
...
>>> f = Foo()
>>> f[1]
10
>>> for i in f: print(i)
0
10
20
>>> 15 in f
False
```

虽然没有 __iter__ 方法，但是 Foo 实例是可迭代对象，因为发现有 __getitem__ 方法时，Python 会调用它，传入从 0 开始的整数所有，尝试迭代对象 (这是一种后备机制)。尽管没有实现 __contains__ 方法，但是 Python 足够智能，能迭代 Foo 实例，因此也能使用 in 运算符：Python 会做全面检查，看看有没有指定的元素。

综上，鉴于序列协议的重要性，如果没有 __iter__ 和 __contains__ 方法，Python 会调用 __getitem__ 方法，设法让迭代和 in 运算符可用。

第 1 章定义的 FrenchDeck 类也没有继承 abc.Sequence，但是实现了序列协议的两个方法：__getitem__ 和 __len__。

```
# 例1-1: 几个内置函数的理解
import collections

Card = collections.namedtuple('Card', ['rank', 'suit'])

class FrenchDeck:
    ranks = [str(n) for n in range(2, 11)] + list('JQKA')
    suits = 'spades diamonds clubs hearts'.split()

```

```
    def __init__(self):
        self._cards = [Card(rank, suit)
                       for suit in self.suits for rank in self.ranks]

    def __len__(self): # 内置函数 len()
        return len(self._cards)

    def __getitem__(self, position): # 内置迭代器
        return self._cards[position]
```

第 1 章那些示例之所以能用，大部分是由于 Python 会特殊对待那些看起来像是序列的对象。Python 中的迭代是鸭子类型的一种极端形式：为了迭代对象，解释器会尝试调用两个不同的方法。

11.3 使用猴子补丁在运行时实现协议

例 1-1 有个重大缺陷：无法洗牌。作者第一次编写 `FrenchDeck` 示例时，实现了 `shuffle` 方法。后来，作者对 Python 风格有了更深刻的理解，发现如果 `FrenchDeck` 实例的行为像序列，那么他就不需要 `shuffle` 方法，因为已经有了 `random.shuffle` 函数可用。

标准库中的 `random.shuffle` 用法如下：

```
>>> from random import shuffle
>>> l = list(range(10))
>>> shuffle(l)
[4, 7, 5, 3, 8, 9, 6, 1, 0, 2]
```

然而，如果尝试打乱 `FrenchDeck` 实例，则会出现异常："'FrenchDeck' object does not support item assignment"。这个问题的原因是，`shuffle` 函数要调换集合中元素的位置，而 FrenchDeck 只实现了不可变的序列协议。可变的序列还必须提供
texttt__setitem__ 方法。

Python 是动态语言，因此我们可以在运行时修正这个问题，甚至还可以在交互式控制台中，修正方法如下：

```
>>> def set_card(deck, position, card):
...     deck._cards[position] = card
...
>>> FrenchDeck.__setitem__ = set_card
>>> shuffle(deck)
```

特殊方法 `__setitem__` 的签名在 Python 语言参考手册中定义，语言参考中使用的参数是 `self,key,value`。这里使用的是 `deck, position, card`。这么做是为了告诉你，每个 Python 方法说到底都是普通函数，把第一个参数命名为 `self` 只是一定约定。在控制台中使用那几个参数没有问题，不过在 Python 源码文件中最好按约定使用 `self,key,value`。

这里的关键是，`set_card` 函数要知道 `deck` 对象有一个名为 `_cards` 的值必须是可变序列，然后，我们把 `set_card` 函数赋值给特殊方法 `__setitem__`，从而把它依附到 `FrenchDeck`

类上。这种技术叫猴子补丁：在运行时修改类或模块，而不该动源代码。猴子补丁很强大，但是打补丁的代码与要打补丁的程序耦合十分紧密，而且往往要处理隐藏和没有文档的部分。

处理举例说明猴子补丁之外，上述示例还强调了协议是动态的：`random.shuffle` 函数不关心参数的类型，只要那个对象是西安了部分可变序列协议即可。即便对象一开始没有所需的方法也没关系，后来再提供也行。

11.4 Alex Martelli 的水禽

这节多为介绍性内容，这里省略了很多，请参考原文第 262 页。

继承抽象基类很简单，只需要实现所需的方法，这样也能明确开发者的意图。此外，使用 `isinstance` 和 `issubclass` 测试抽象基类更为人接受。

然而，即便是抽象基类，也不能滥用 `isinstance` 检查，用的多了可能导致代码异味，即表明面向对象设计得不好。在一连串 `if/elif/elif` 中使用 `isinstance` 做检查，然后根据对象的类型执行不同的操作，通常是不好的做法；此时应该使用多态，即采用一定的方式定义类，让解释器把调用分派给正确的方法。

另一方面，如果必须强制执行 API 契约，通常可以使用 `isinstace` 检查抽象基类。

Alex 多次强调，要抑制住创建抽象基类的冲动。滥用抽象基类会造成灾难性的后果，表明语言太注意表面形式，这对以实用和务实著称的 Python 可不是好事。

11.5 定义抽象基类的子类

我们将遵循 Martelli 的建议，先利用现有的抽象基类然后再斗胆自己定义。下面我们将 `FrenchDeck2` 声明为 `collections.MutableSequence` 的子类。

```
# 例11-8: 继承自 MutableSequence 抽象基类的 FrenchDeck2
import collections

Card = collections.namedtuple('Card', ['rank', 'suit'])


class FrenchDeck2(collections.MutableSequence):
    ranks = [str(n) for n in range(2, 11) + list('JQKA')]
    suits = 'spades diamonds clubs hearts'.split()

    def __init__(self):
        self._cards = [Card(rank, suit)
                       for suit in self.suits for rank in self.ranks]

    def __len__(self):
        return len(self._cards)

    def __getitem__(self, position):
        return self._cards[position]
```

```
20
21      def __setitem__(self, position, value): # 支持洗牌所必需的内置函数
22          self._cards[position] = value
23
24      def __delitem__(self, position): # 超类的抽象方法，必须实现
25          del self._cards[position]
26
27      def insert(self, position, value): # 超类的抽象方法，必须实现
28          self._cards.insert(position, value)
```

导入模块时，Python 不会检查抽象方法的实现，在运行时实例化 `FrenchDeck2` 类时才会真正检查。因此，如果没有正确实现某个抽象方法，Python 会抛出 `TypeError` 异常，并把错误消息设置为 “Can’t instantiate abstract class FrenchDeck2 with abstract method __delitem()__, insert” 正是这个原因，即便 `FrenchDeck2` 类不需要 `__delitem__` 和 `insert` 提供的形为，也要实现，因为 `MutableSequence` 抽象基类需要它们。

在 `collections.abc` 中，每个抽象基类的具体方法都是作为类的公开接口实现的，因此不用知道实例的内部结构。

11.6 标准库中的抽象基类

自 Python2.6 开始,在标准库中提供了抽象基类,其中大部分抽象基类在 `collections.abc` 模块中。

11.6.1 `collections.abc` 模块中的抽象基类

标准库中有两个名为 `abc` 的模块，这里说的是 `collections.abc`。为了减少加载时间，Python3.4 在 `collections` 包之外实现了这个模块，因此要与 `collections` 模块分开导入。另一个 `abc` 模块就是 `abc` ，这里定义的是 `abc.ABC` 类，每个抽象基类都依赖于这个类，但是不用导入它，除非定义新抽象基类。

Python3.4 在 `collections.abc` 模块中定义了 16 个抽象基类，其继承关系如下图。

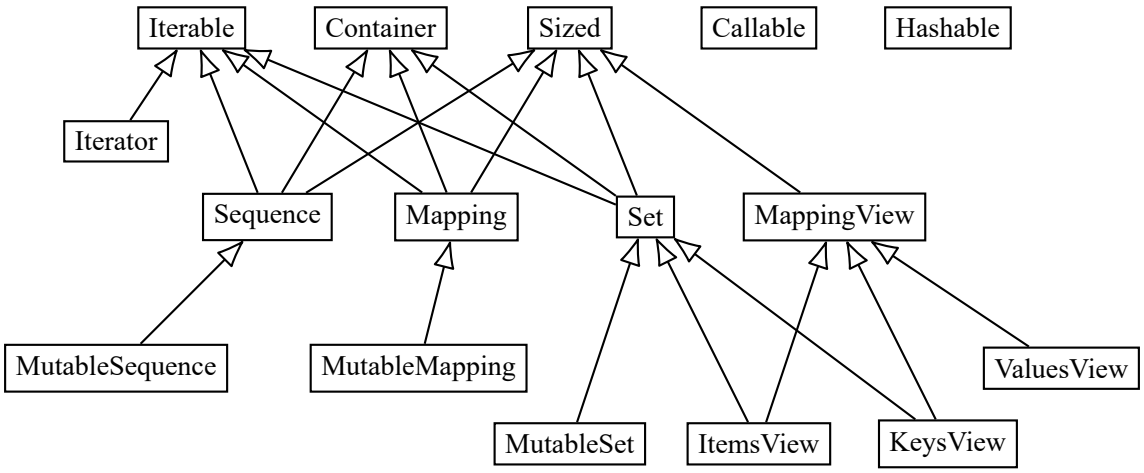

图 11.2 collections.abc 抽象基类继承关系

下面详述这些基类：

- `Iterable`，`Container`，`Sized`

  各个集合应该继承自这三个抽象基类，或者至少实现兼容的协议。

  - `Iterable` 通过 `__iter__` 方法支持迭代。
  - `Container` 通过 `__contains__` 方法支持 `in` 运算符。
  - `Sized` 通过 `__len__` 方法支持 `len()` 函数。

- `Sequence`，`Mapping`，`Set`

  这三个是主要的不可变集合类型，而且各自都有可变的子类。

- `MappingView`

  在 Python3 中，映射方法 `.items()`, `.keys()`, `.values()` 返回的对象分别是 `ItemsView`, `KeysView`, `ValuesView` 的实例。前两个类还从 Set 类继承了丰富的接口。

- `Callable`，`Hashable`

  这两个抽象基类与集合没有太大关系，只不过因为 `collections.abc` 是标准库中定义抽象基类的第一个模块，而它们又太重要了，因此才把它们放在了 `collections.abc` 模块中。原作者从未见过 `Callable`，`Hashable` 的子类。这两个抽象基类的主要作用是为内置函数 `isinstance` 提供支持，以一种安全的方式判断对象能不能调用或散列。若想检查是否能调用，可以使用内置的 `callable()` 函数，但是没有类似的 `hashable()` 函数，因此测试对象是否可散列，最好使用 `isinstance(my_obj,Hashable)`。

- `Iterator`

  注意它是 `Iterable` 的子类，会在后文中详细讨论。

11.6.2 抽象基类的数字塔

`numbers` 包定义的是 “数字塔”(即各个抽象基类的层次结构是线性的)，其中 `Number` 是位于最顶端的超类，随后是 `Complex` 子类，依次往下，最底端的是 `Integral` 类[9]。

[9]包括 `Number, Complex, Real, Rational, Integral`

因此，如果想检查一个数是不是整数，可以使用 `isinstance(x, numbers.Integral)`，这样代码就能接受 `int`，`bool`，或者外部库使用 `numbers` 抽象基类注册的其他类型。

与之类似，如果一个值可能是浮点数类型，可以使用 `isinstance(x, numbers.Real)` 检查。这样代码就能接受 `bool`，`int`，`float`，`fraction.Fraction` 或者外部库提供的非复数类型。

11.7 定义并使用一个抽象基类

假设我们在构建一个广告管理框架，名为 ADAM。它的职责之一是，支持用户提供随机挑选的无重复类。为了让 ADAM 的用户明确理解 “随机挑选的无重复” 组件是什么意思，我们将定义一个抽象基类。

我们将这个抽象基类命名为 `Tombola`，它有四个方法，其中两个是抽象方法：

- `.load(...)`: 把元素放入容器。
- `.pick()`: 从容器中随机拿出一个元素，返回选中的元素。

另外两个是具体方法：

- `.loaded()`: 如果容器中至少有一个元素，返回 `True`。
- `.inspect()`: 返回一个有序元组，由容器中的现有元素构成，不会修改容器的内容 (内部顺序不保留)。

下图展示了 Tombola 抽象基类和三个具体实现：

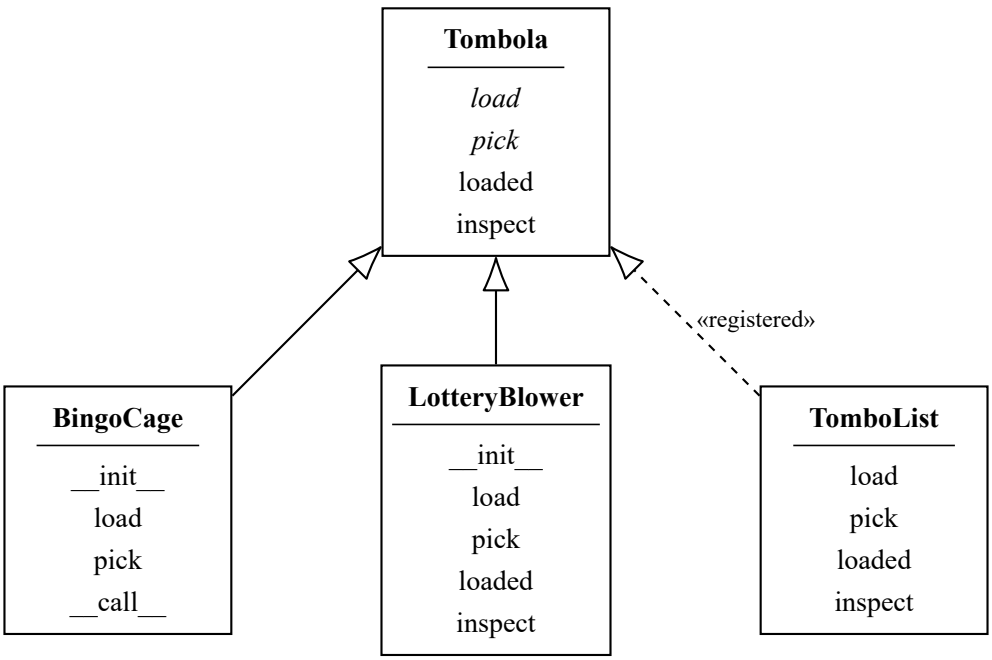

图 11.3 Tombola 继承关系

Tombola 抽象基类的定义如下：

```
1 # 例11-9: Tombola
```

```
import abc

class Tombola(abc.ABC):  # 自定义抽象基类需要继承自 abc.ABC

    @abc.abstractmethod  # 抽象方法装饰器
    def load(self, iterable):
        """从可迭代对象中添加元素"""

    @abc.abstractclassmethod
    def pick(self):
        """随机删除元素，然后将其返回
        如果实例为空，这个方法应该抛出 'LookupError'
        """

    def loaded(self):    # 抽象基类可以包含具体方法
        """如果至少有一个元素，返回 'True'，否则返回 'False'"""
        return bool(self.inspect())  # 抽象基类中的具体方法只能依赖抽象基类定义的接口

    def inspect(self):
        """返回一个有序元组，由当前元素构成"""
        items = []
        while True:
            try:
                items.append(self.pick())
            except LookupError:
                break
        self.load(items)  # 挑选完后，再用 .load 把元素放回去
        return tuple(sorted(items))
```

抽象基类可以有实现方法，但即便实现了，子类也必须覆盖抽象方法，但是在子类中可以使用 `super()` 函数调用抽象方法，为它添加功能，而不是从头开始实现。

在上述实例中，`.inspect()` 方法有点笨拙，这只是为了强调抽象基类可以提供具体方法，只要依赖接口中的其他方法就行。

11.7.1 抽象基类句法详解

声明抽象基类最简单的方式是继承 `abc.ABC` 或其他抽象基类。然而，`abc.ABC` 是 Python3.4 新增的类，因此如果使用旧版的 Python 并且不能继承现有的抽象基类，必须在 `class` 语句中使用 metaclass = 关键字，把值设为 `abc.ABCMeta`：

```
    class Tombola(metaclass = abc.ABCMeta):
    # ...
```

如果实在更遥远的 Python2 版本，则需要使用 `__metaclass__` 类[10]属性。

```
    class Tombola(object):
    __metaclass__ = abc.ABCMeta
    # ...
```

[10]元类会在后面章节讲解

除了 @abstractmethod 之外，abc 模块还定义了 @abstractclassmethod, @abstractstaticmethod, @abstractproperty 三个修饰器，但这三个装饰器在 Python3.3 起废弃了，因为装饰器可以在 @abstractmethod 上堆叠，这三个就显得多余了。例如，声明抽象类方法的推荐方式是：

```
    class MyABC(abc.ABC):
    @classmethod
    @abc.abstractmethod
    def an_abstract_method(cls,...):
    pass
```

11.7.2 定义 Tombola 抽象基类的子类

下面我们来定义 Tombola 的抽象子类：BingoCage。

```
# 例11-14: BingoCage
import random
from tombola import Tombola

class BingoCage(Tombola):
    def __init__(self, items):
        self._randomizer = random.SystemRandom()
        self._items = []
        self.load(items)

    def load(self, items):
        self._items.extend(items)
        self._randomizer.shuffle(self._items)

    def pick(self):
        try:
            return self._items.pop()
        except IndexError:
            raise LookupError('pick from empty BingoCage')

    def __call__(self):
        self.pick()
```

下面我们再定义一个抽象子类：LotteryBlower，它对两个耗时的 loaded 和 inspect 方法进行了重写。

```
# 例11-14: LotteryBlower
import random
from tombola import Tombola

class LotteryBlower(Tombola):

    def __init__(self, iterable):
        self._balls = list(iterable) # 存入属性，而不是引用

    def load(self, iterable):
        self._balls.extend(iterable)
```

```
12
13      def pick(self):
14          try:
15              position = random.randrange(len(self._balls)) # 如果长度为 0 抛出 ValueError
16          except ValueError:
17              raise LookupError('pick from empty LotteryBlower')
18          return self._balls.pop(position)
19
20      def loaded(self):
21          return bool(self._balls)
22
23      def inspect(self):
24          return tuple(sorted(self._balls))
```

11.7.3 `Tombola` 的虚拟子类

白鹅类型的一个基本特征：即便不继承，也有办法把一个类注册为抽象基类的虚拟子类。这样做时，我们保证注册的类忠实地实现了抽象基类定义的接口，而 Python 会相信我们，从而不做检查。如果我们说谎了，常规的运行时异常会被我们捕获。

注册虚拟子类的方式是再抽象基类上调用 register 方法。这么做之后，注册的类会变成抽象基类的虚拟子类，而且 issubclass 和 isinstance 等函数都能识别，但是注册的类不会从抽象基类中继承任何方法或属性。

虚拟子类不会继承注册的抽象基类，而且任何时候都不会检查它是否符合抽象基类的接口，即便再实例化时也不会检查。为了避免运行时粗偶，虚拟子类要实现所需的全部方法。

`register` 方法通常作为普通的函数调用，不过也可以作为装饰器使用。下面我们将实现 `TomboList` 类，其继承关系如下图：

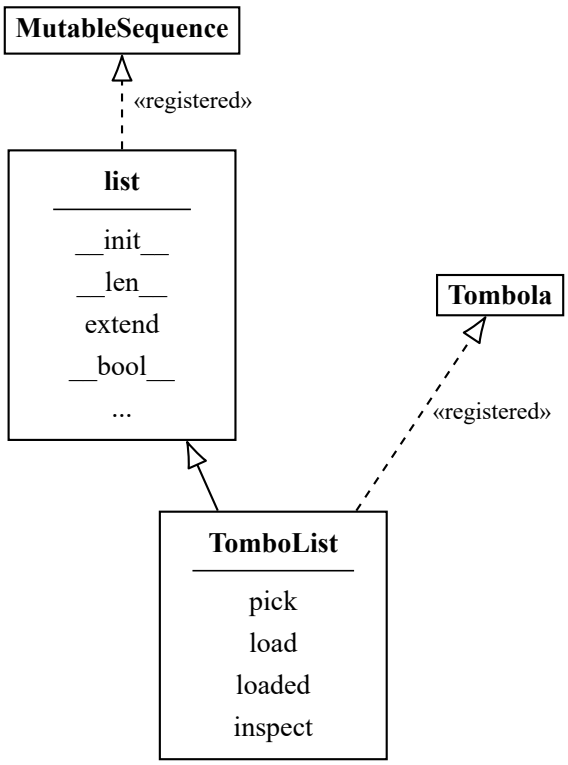

图 11.4 TomboList 继承关系

下面我们将使用装饰器语法实现 `TomboList` 类：

```
# 例11-14: TomboList
from random import randrange
from tombola import Tombola

@Tombola.register     # 使用装饰器注册为 Tombola 的子类
class TomboList(list):

    def pick(self):
        if self:       # 从 list 中继承 __bool__ 方法，列表不为空时返回 True
            position = randrange(len(self))
            return self.pop(position)
        else:
            raise LookupError('pop from empty TomboList')

    load = list.extend # TomboList.load 与 list.extend 一样

    def loaded(self):
        return bool(self)

    def inspect(self):
        return tuple(sorted(self))

# Tombola.register(TomboList) Python3.3 之前的写法
```

注册之后，可以使用 `issubclass` 和 `isinstance` 函数判断 `TomboList` 是不是 `Tombola` 的子类：

```
>>> issubclass(TomboList, Tombola)
True
>>> t = TomboList(range(100))
>>> isinstance(t, Tombola)
True
```

类的继承关系在一个特殊的类属性中指定 `__mro__`，即方法解析顺序。这个属性的作用很简单，按顺序列出类及其超类，Python 会按照这个顺序搜索方法。查看 `TomboList` 类的 `__mro__` 属性，你会发现它只列出了“真实的”超类，即 `list` 和 `object`。

```
>>> TomboList.__mro__
(<class 'tombolist.TomboList'>, <class 'list'>, <class 'object'>)
```

`TomboList.__mro__` 中没有 `Tombola`，因此 `TomboList` 没有从 `Tombola` 中继承任何方法。

11.8 Tombola 子类的测试方法

下面介绍两个用于测试的方法：

- `__subclass__()`
  这个方法返回类的直接子类列表，不含虚拟子类。
- `_abc_registry`
  只有抽象基类有这个数据属性，其值是一个 `WeakSet` 对象，即抽象类注册的虚拟子类的弱引用。

测试脚本过长，这里不给出，读者可自行查阅原文 P278 页。

11.9 Python 使用 register 的方法

示例 11-14 将 Tombola.register 当作类装饰器使用。但在 Python3.3 之气的版本中不能这样使用，必须再定义类后像普通函数那样调用。

虽然现在可以把 `register` 当作装饰器使用，但更常见的做法还是当作函数使用，用于注册其他地方定义的类。例如 `Sequence`：

```
Sequence.register(tuple)
Sequence.register(str)
Sequence.register(range)
Sequence.register(memory)
```

11.10 鹅的行为有可能像鸭子

即便不注册，抽象基类也能把一个类识别为虚拟子类。

```
>>> class Struggle:
...     def __len__(self): return 23
...
>>> from collections import abc
>>> isinstance(Struggle(), abc.Sized)
True
>>> issubclass(Struggle, abb,Sized)
True
```

可以发现，`Struggle` 没有继承自 `abc.Sized` 但却是它的子类，这是因为 `abc.Sized` 实现了一个特殊的类方法，名为 `__subclasshook__`。

```
# 例11-17: Sized 类部分源码
from abc import ABCMeta, abstractmethod

class Sized(metaclass = ABCMeta):

    __slots__ = ()

    @abstractmethod
    def __len__(self):
        return 0

    @classmethod
    def __subclasshook__(cls,C):
        if cls is Sized:
            if any("__len__" in B.__dict__ for B in C.__mro__):
                return True    # 如果超类有 __len__ 返回 True
        return NotImplemented # 否则，返回 NotImplemented 让子类检查
```

原作者并不建议程序员自己实现 `__subclasshook__` 方法，原因是可靠性很低。

12 继承的优缺点

12.1 子类化内置类型很麻烦

Python 允许内置类型子类化，但有个重要的注意事项：内置类型不会调用用户定义的类覆盖的特殊方法。

```
# 例12-1: 内置类型子类化
class DoppelDict(dict):
    def __setitem__(self, key, value):
        super().__setitem__(key,[value] * 2)

dd = DoppelDict(one = 1) # {'one': 1}
dd['two'] = 2            # {'one': 1, 'two': [2, 2]}
dd.update(three = 3)     # {'one': 1, 'two': [2, 2], 'three': 3}
```

在上述代码中，只有直接调用 `__setitem__` 的 `dd['two'] = 2` 代码返回了我们所期望的指。而其他间接调用 `__setitem__` 的方法都忽略了我们覆盖的 `__setitem__` 方法。不仅如此，内置类型的方法调用的其他类方法，如果被覆盖了，也不会被调用。

原生类型的这种行为违背了面向对象编程的一个基本原则：始终应该从实例 (`self`) 所属的类开始搜索方法，即使在超类实现的类中调用也是如此。在这种糟糕的局面中，`__missing__` 方法却能按预期方法工作，不过这只是特例。

直接子类化内置类型容易出错，因为内置类型的方法通常会忽略用户覆盖的方法。用户自定义的类应该继承自 `collections` 模块中的类，这些类做了特殊设计，因此易于扩展。

下面是一个继承自 `collections.UserDict` 的例子：

```
# 例12-3: 继承自 UserDict 的字典子类
import collections

class DoppelDict2(collections.UserDict):
    def __setitem__(self, key, value):
        super().__setitem__(key, [value] * 2)

dd = DoppelDict2(one = 1) # {'one': [1, 1]}
dd['two'] = 2             # {'one': [1, 1], 'two': [2, 2]}
dd.update(three = 3)      # {'one': [1, 1], 'two': [2, 2], 'three': [3, 3]}
```

12.2 多重继承和方法解析顺序

任何实现多重继承的语句都要处理潜在的命名冲突，这种冲突由不相关的祖先类实现同名方法引起。我们看下面这个例子。

```
# 多继承的命名冲突
class A:
    def ping(self):
```

```
        print('ping:', self)

class B(A):
    def pong(self):
        print('pong:',self)

class C(A):
    def pong(self):
        print('PONG:',self)

class D(B, C):
    def ping(self):
        print('post-ping:',self)

    def pingpong(self):
        self.ping()
        super().ping()
        self.pong()
        super().pong()
        C.pong(self)
```

下面我们在 D 的实例上调用 d.pong() 方法：

```
>>> d = D()
>>> d.pong()  # 直接调用 pong 运行的是第一个继承的 B 版本。
pong:<diamond.D object at 0x......>
>>> C.pong(d) # 超类中的方法都可以直接调用，此时要把实例作为显式参数传入。
PONG:<diamond.D object at 0x......>
```

Python 能区分 d.pong() 调用的是哪个方法，是因为 Python 会按照特定的顺序遍历继承图。这个顺序叫方法解析顺序。类都有一个名为 __mro__ 的属性，它的值是一个元组，按照方法解析顺序列出各个超类，从当前类一直向上，直到 object 类。D 类的 __mro__ 属性如下：

```
>>> D.__mro__
(<class 'diamond.D'>, <class 'diamond.B'>, <class 'diamond.C'>, <class 'diamond.A'>, <class
    'object'>)
```

若想把方法调用委托给超类，推荐的方式是使用内置的 super() 函数。有时可能要绕过方法解析顺序，直接调用某个超类的方法，这样做有时更方便。例如 D.ping 方法可以这样写：

```
def ping(self):
    A.ping(self)  # 不是 super().ping()
    print('posy-ping:', self)
```

注意，直接在类上调用实例方法时，必须显式传入 self 参数，因为这样访问的是未绑定方法。但是最好还是使用 super() 方法，这样既安全，又不易过时。尤其是在调用框架或不受自己控制的类层次结构时。

下面来看在 D 上调用 pingpong 方法得到的结果。

```
>>> d = D()
>>> d.pingpong()
post-ping: <__main__.D object at 0x00000207D6E80148>
ping: <__main__.D object at 0x00000207D6E80148> # 调用超类 A 中的 ping() 方法
pong: <__main__.D object at 0x00000207D6E80148> # 自己没有，向上找到 B 的 pong() 方法
pong: <__main__.D object at 0x00000207D6E80148> # 向上找到 B 的 pong() 方法
PONG: <__main__.D object at 0x00000207D6E80148> # 忽略 __mro__，直接调用 C 中的 pong() 方法
```

方法解析顺序不仅考虑继承图，还考虑子类声明中列出超类的顺序。也就是说 `class D(B, C)` 会先搜索 B 再搜索 C。在内部实现中，方法解析顺序使用 C3 算法，不过除非大量使用复杂的多继承，否则无需了解。

12.3 处理多重继承

在日常使用中，虽然多继承没有缺陷，但是应该尽量避免使用多继承。多继承会增加复杂度，往往会导致难以理解的错误。下面是避免多继承混乱的一些建议。[11]

1. 把接口继承可实现继承区分开

   使用多重继承时，一定要明确一开始为什么创建子类：

   - 继承接口：创建子类时，实现 “是什么” 关系。
   - 继承实现：通过重用避免代码重复。

2. 使用抽象基类显式表示接口

   如果类的作用是定义接口，应该明确把它定义为抽象基类。

3. 通过混入重用代码

   如果一个类的作用是为了多个不相关的子类提供方法实现，从而实现重用，但不体现 “是什么” 关系，应该把那个类明确定义为混入类。从概念上讲，混入不定义新类型，只是打包方法，便于重用。混入类绝对不能实例化，而且具体类不能只继承混入类。混入类应该提供某方面的特定行为，只实现少量关系非常密切的方法。

4. 在名称中明确指明混入

   Python 并没有把类声明为混入的正规方式，所以强烈建议在名称中加入 `...Mixin` 后缀表示混入类。在 UML 中使用 *«mixin»* 标记。

5. 抽象基类可以作为混入，反过来则不成立

   抽象基类可以实现具体方法，因此可以作为混入使用。不过，抽象基类会定义类型，而混入做不到。此外，抽象基类可以作为其他类的唯一基类，而混入绝不能作为唯一的超类，除非继承另一个更具体的混入 (实际上很少这样用)。

6. 不要子类化多个基类

   具体类可以没有，或者最多只有一个具体超类。也就是说，具体类的超类中除了这一个具体超类之外，其余的都是抽象基类或混入。

7. 为用户提供聚合类

[11] 这里跳过了原书 12.3 节，这节讲解的是 Tkinter 的多继承实例

如果抽象基类或混入的组合对客户代码非常有用，那就提供一个类，使用易于理解的方式把它们结合起来。

8. 优先使用对象组合，而不是类继承

优先使用组合能让设计更灵活。子类化是一种紧耦合，而且较高的继承树容易倒。

13 正确重载运算符

本章只讨论一元运算符和中缀运算符。[12]

13.1 运算符重载基础

在某些圈子中，运算符重载的名声并不好。比如 Java 禁止用户重载运算符。对此 Python 施加了一些限制，做好了灵活性，可用性和安全性方面的平衡：

- 不能重载内置类型的运算符
- 不能新建运算符，只能重载现有的运算符
- 某些运算符不能重载：`is`, `and`, `or`, `not`

13.2 一元运算符

在 Python 语言参考手册中，列出了三个一元运算符。下面是这三个运算符和对应的特殊方法：

- `-` (`__neq__`)
  一元负数运算符。如果 `x` 是 `-2`，那么 `-x == 2`
- `+` (`__pos__`)
  一元取正算术运算符。通常[13] `x == +2`
- `~`(`__invert__`)
  对整数按位取反，定义 `~x == -(x+1)`

除此之外，还有 `abs()` 函数也被列为了一元运算符。前面已多次提及。

支持一元运算符非常简单，只需实现相应的特殊方法。这些特殊方法只有一个参数：`self`。不过，要遵守运算符的一个基本规则：始终返回一个新对象。也就是说，不能修改 `self`，要创建并且返回新的实例。

下面给出了之前定义的 `Vector` 类实现的几个新运算符：

```
def __abs__(self):
    return math.sqrt(sum(x*x for x in self))

def __neg__(self):
    return Vector(-x for x in self)

def __pos__(self):
    return Vector(self)   # 注意：即使相同，也要构建新的实例。
```

[12] 其他运算符如函数调用，属性访问等大多不可重载，用户也不能修改用法。

[13] 例外：`Decimal` 大整数运算，`collections.Counter` 计算运算。原书 P309 有说明。

13.3 重载向量加法运算符 +

两个欧几里得向量加在一起得到的是一个新的向量，它的各个分量是两个向量中相应的分量之和，如果长度不同，则应对部分元素进行相加：

下面在之前定义过的 Vector 类基础上添加加法运算：

```
def __add__(self, other):
    pairs = itertools.zip_longest(self, other, fillvalue = 0.0) # 生成一个 (a,b) 形式的元组
    return Vector(a + b for a, b in pairs) # 返回新的实例
```

这里我们使用了 zip_longest 函数,它能处理任何可迭代对象,但仅当左操作数是 Vector 实例时才有效。

```
>>> v1 = Vector([3,4,5])
>>> v1 + (10, 20, 30)
Vector([13.0, 24.0, 35.0])
>>> (10, 20, 30) + v1
Traceback (most recent call last):.....
```

为了支持涉及不同类型的运算，Python 为中缀运算符特殊方法提供了特殊的分派机制。对 a+b 来说，解释器会提供以下几步操作：

- 如果 a 有 __add__ 方法，而且返回值不是 NotImplemented，调用 a.__add__(b)，然后返回结果。
- 如果 a 没有 __add__ 方法，或者返回值是 NotImplemented，检查 b 有没有 __radd__ 方法，如果有，而且返回值不是 NotImplemented，调用 b.__radd__(a)，然后返回结果。
- 如果 b 没有 __radd__ 方法，或者返回值是 NotImplemented，抛出 TypeError。

__radd__ 是 __add__ 的“反射”，“右向”版本。以 r 开头的内置方法多为类似功能。

因此，如果我们要实现 Vecotr 实例的 (10, 20, 30) + v1 方法，只需实现 __radd__ 内置函数即可。

```
def __add__(self, other):
    pairs = itertools.zip_longest(self, other, fillvalue = 0.0)
    return Vector(a + b for a, b in pairs)

def __radd__(self, other):
    return self + other   # 直接委托给 __add__
```

在实现了右向运算后，我们又遇到一个问题：如果提供的对象不可迭代，那么 __add__ 就无法处理,此时如果我们强行运算,就会抛出一堆难以理解的错误。对此 Python 有一个规定：如果由于类型不兼容而导致运算符特殊方法无法返回有效的结果,那么应该返回 NotImplemented，而不是抛出 TypeError。返回 NotImplemented 时，另一个操作数所属的类型还有机会执行运算，即 Python 会尝试调用反向方法。

下面是 Vector 加法运算的最终版：

```
def __add__(self, other):
    try:
        pairs = itertools.zip_longest(self, other, fillvalue = 0.0)
        return Vector(a+b for a,b in pairs)
    except TypeError:
        return NotImplemented
```

13.4 重载标量乘法运算符 *

在 Vector 实例中使用乘法运算：Vector([1, 2, 3]) * x。如果 x 是数字，就是计算标量积，结果是一个新 Vector 实例，各个分量都会乘以 x ，这也叫元素级乘法。

涉及乘法运算的还有点积；Numpy 等库目前的做法是，不提供两种意义的 * 运算，而是单独使用 dot() 函数计算点积。

下面我们实现简单的乘法运算：

```
def __mul__(self, scalar):
    return Vector(n * scalar for n in self)

def __rmul__(self, scalar):
    return self * scalar
```

这里有个问题，scalar 必须是实数，我们可以像加法运算那样使用 try 语句，但有个更易于理解的方法。我们将使用 isinstance() 检查 scalar 的类型，但是不硬编码具体的类型，而是检查 numbers.Real 抽象基类的真实子类或虚拟子类的数值类型。

```
from array import array
import reprlib
import math
import functools
import operator
import itertools
import numbers    # 为了检查类型导入

class Vector:
    typecode = 'd'

    def __init__(self, components):
        self._components = array(self.typecode, components)

    def __mul__(self, scalar):
        if isinstance(scalar, numbers.Real):    # 检查是否是 numbers.Real 的子类实例
            return Vector(n * scalar for n in self)
        else:
            return NotImplemented

    def __rmul__(self, scalar):
        return self * scalar

```

```
    #......
```

这样实现后，Vector 实例就可以与各种实数进行乘法运算了。

通过 + 和 * 运算，我们了解了中缀表达式的常用模式，其相关的计数如下表所示：

表 4.1 中缀运算符方法名

运算符	正向方法	反向方法	就地方法	说明
+	__add__	__radd__	__iadd__	加法或拼接
-	__sub__	__rsub__	__isub__	减法
*	__mul__	__rmul__	__imul__	乘法或重复复制
/	__truediv__	__rtruediv__	__itruediv__	除法
//	__floordiv__	__rfloordiv__	__ifloordiv__	整除
%	__mod__	__rmod__	__imod__	取余
divmod()	__divmod__	__rdivmod__	__idivmod__	商和模组成的元组
**, pow()	__pow__	__rpow__	__ipow__	幂运算
@	__matmul__	__rmatmul__	__imatmul__	矩阵乘法
&	__and__	__rand__	__iand__	位与
|	__or__	__ror__	__ior__	位或
^	__xor__	__rxor__	__ixor__	位异或
<<	__lshift__	__rlshift__	__ilshift__	按位左移
>>	__rshift__	__rrshift__	__irshift__	按位右移

13.5 众多比较运算符

Python 解释器对众多比较运算符的处理与前文类似，不过在两个方面又重大区别：

- 正向和反向调用使用的是同一系列方法。
- 对 == 和 != 来说，如果反向调用失败，Python 会比较对象的 ID，而不是抛出 TypeError。

表 4.2 比较运算符方法名

分组	运算符	正向方法调用	反向方法调用	后备机制
相等性	a==b	a.__eq__(b)	b.__eq__(a)	返回 id(a) == id(b)
	a!=b	a.__ne__(b)	b.__ne__(a)	返回 not(a == b)
排序	a>b	a.__gt__(b)	b.__lt__(a)	抛出 TypeError
	a<b	a.__lt__(b)	b.__gt__(a)	抛出 TypeError
	a>=b	a.__ge__(b)	b.__le__(a)	抛出 TypeError
	a<=b	a.__le__(b)	b.__ge__(a)	抛出 TypeError

之前我们是这样实现 Vector.__eq__ 方法的：

```
def __eq__(self, other):
    return (len(self) == len(other) and all(a == b for a,b in zip(self, other)))
```

这样做会有一个缺陷：会导致 Vector 实例能与所有的可迭代对象进行比较，究竟是否要这样做，要视情况二定。但对操作数过度宽容会可能导致令人惊讶的结果，而程序员讨厌惊喜。

因此我们应该更保守一点：

```
def __eq__(self, other):
    if isinstance(other, self):  # 如果是 Vector 或者其子类实例
        return (len(self) == len(other) and all(for a,b in zip(self, other)))
    else:
        return NotImplemented     # 如果不是，交给反向比较判断
```

值得说明的是，我们一般不需要自定义 != 于是暖，因为从 object 继承的 __ne__ 方法后备行为满足了我们的需求：定义了 __eq__ 方法，而且它不返回 NotImplemented，__ne__ 方法会对 __eq__ 方法返回的结果取反。当然如果对某些特定运算，需要用户自己实现两种方法，而不是简单取反。

13.6 增量运算符

在外面实现 +,* 运算后，Vector 类已经支持增量运算符 +=,*= 了：

```
>>> v1 = Vector([1,2,3])
>>> v1_alias = v1         # 复制一份，后面审查啊
>>> id(v1)
4302860128
>>> v1 += Vector([4,5,6])
>>> v1                    # 增量加法运算，结果与预期相符
Vector([5.0,7.0,9.0])
>>> id(v1)                # id 发生了变化，创建了新实例
4302859904
>>> v1_alias              # 原来的 Vector 没有变
Vector([1.0,2.0,3.0])
>>> v1*=11
>>> v1                    # 增量乘法运算，结果与预期相符
Vector([55.0,77.0,99.0])
>>> id(v1)                # 创建了新实例
4302858336
```

如果一个类没有实现就地运算，增量赋值运算符只是语法糖: a+=b 的作用与 a=a+b 完全一样。对不可变类型来说，这是预期行为，而且，如果定义了 __add__ 方法，就没有必要写格外的代码。

然而，如果实现了就地运算方法，计算的结果会调用对应方法。这种运算符的名称表明，它们会就地修改左操作数，而不会创建新的对象作为结果。对于不可变了类型，一定不能实

现就地运算。

如果是可迭代对象，往往需要实现 `__iadd__` 这类的方法。并且，通过观察 `list` 方法的实现，我们可以知道，与 `+` 相比，`+=` 运算符对第二个操作数更宽容。`+` 运算符的两个操作数必须是相同类型，如若不然，结果的类型可能让人摸不着头脑。而 `+=` 的情况更明确，因为就地修改左操作数，所以结果的类型是确定的。

下面我们使用 `BingoCage` 的子类，演示 `+=` 运算符的实现：

```
class AddableBingoCage(BingoCage):

    def __add__(self, other):
        if isinstance(self, Tombola): # 第二个操作数只能是 Tombola 及其子类
            return AddableBingoCage(self.inspect() + other.inspect())
        else:
            return NotImplemented

    def __iadd__(self, other):
        if isinstance(other, Tombola):
            other_iterable = other.inspect()
        else:
            try:
                other_iterable = iter(other) # iter 将在下一章讨论
            except TypeError:                # 若失败，告诉用户该怎么做
                self_cls = type(self).__name__
                msg = "right operand in += must be {!r} or an iterable"
                raise TypeError(msg.format(self_cls))
        self.load(other_iterable) # 如果能执行到这里，载入
        return self    # 增量赋值必须返回 self
```

此外，我们也无需实现 `__radd__` 方法，因为不需要。如果右操作数是相同类型，那么正向方法 `__add__` 会处理。

V 控制流程

14 可迭代对象，迭代器和生成器

Python 中，所有的生成器都是迭代器，因为生成器完全实现了迭代器接口。并且，大多数情况下，迭代器和生成器都被视作同一概念。

在 Python3 中，生成器具有广泛的作用。即使是内置的 `range()` 函数也返回一个类似生成器的对象，而以前则返回完整的列表。如果一定要返回列表，那么必须明确指出 (如：`list(range(10))`)。

在 Python 中，所有集合都是可迭代的。在 Python 语言内部，迭代器用于支持：

- `for` 循环。
- 构建和扩展集合类型
- 逐行遍历文本文件
- 列表推导，字典推导和集合推导
- 元组拆包
- 调用函数时，使用 * 拆包实参

14.1 Sentence 类第 1 版：单词序列

我们要实现一个 `Sentence` 类，以此打开探索可迭代对象的旅程。我们向这个类的构造函数传入包含一些文本的字符串，然后可以逐个单词迭代。第 1 版要实现序列协议：

```
# 例14-1: Sentence 第一版
import re
import reprlib

RE_WORD = re.compile('\w+')  # 匹配 ASCII 组成的单词


class Sentence:
    def __init__(self, text):
        self.text = text
        self.words = RE_WORD.finall(text) # 满足正则表达式的全部非重复匹配

    def __getitem__(self, index):
        return self.words[index]

    def __len__(self):
        return len(self.words)

```

```
    def __repr__(self):
        return 'Sentence (%s)' % reprlib.repr(self.text) # 限制长度，默认 30
```

序列可迭代的原因：iter 函数

解释器需要迭代对象 x 时，会自动调用 iter(x)。内置的 iter 函数有以下作用：

1. 检查对象是否实现了 __iter__ 内置函数，如果实现了则调用，获取一个迭代器。
2. 如果没有 __iter__ 方法，但是有 __getitem__ 方法，则创建一个迭代器，尝试按顺序 (索引 0 开始) 获取元素。
3. 如果尝试失败，则抛出错误：TypeError：“xx object is not iterable”。

任何序列类型都可迭代，是因为它们都实现了 __getitem__ 内置方法。其实，标准的序列都实现了 __iter__ 内置方法，因此我们也应该这么做。之所以对 __getitem__ 方法作特殊处理，是为了向后兼容，未来可能不会再这么做。

在鸭子类型中，只要实现特殊的 __iter__ 方法，或者实现 __getitem__ 方法且 __getitem__ 方法的参数是从 0 开始的整数，就可以认为对象是可迭代的。

在白鹅类型理论中，可迭代对象的定义简单一些，不过没那么灵活：如果实现了 __iter__ 方法，那么就认为对象是可迭代的。此时，不需要创建子类，也不用注册，因为 abc.Iterable 类实现了 __subclasshook__ 方法，下面举个例子：

```
>>> class Foo:
...     def __iter__(self):
...         pass
>>> from collections import abc
>>> issubclass(Foo, abc.Iterable)
True
>>> f = Foo()
>>> isinstance(f, abc.Iterable)
True
```

不过要注意，虽然前面定义的 Sentence 类是可迭代的，但却无法通过 issubclass(Sentence, abc.Iterable) 测试。

从 Python3.4 开始，检查对象 x 是否迭代，最准确的方法是调用 iter(x) 函数，如果不可迭代，再处理 TypeError 异常。遮蔽 isinstance(x,abc.Iterable) 更为准确，因为 iter(x) 会考虑到遗留的 __getitem__ 方法。

14.2 可迭代对象与迭代器的对比

两者之间的关系：Python 从可迭代对象中获取迭代器。

下面我们看一个简单的例子：字符串是可迭代对象，背后是有迭代器的，只是我们看不到：

```
>>> s = 'ABC'
>>> for char in s:
```

```
...     print(char)
```

如果没有 for 语句，不自动构建迭代器，我们用 while 语句得这样写：

```
>>> s = 'ABC'
>>> it = iter(s)  # 使用可迭代对象构建迭代器
>>> while(True):
...     try:
...         print(next(it))   # 不断获取下一个字符
...     except StopIteration: # 没有字符了，报错
...         del it            # 释放迭代器对象
...         break             # 退出循环
```

StopIteration 异常表明迭代器到头了。Python 语言内部会处理 for 循环和其他迭代上下文中的 StopIteration 异常。

标准得迭代器接口有两个方法。

- __next__

  返回下一个可用的元素，如果没有，抛出 StopIteration 异常。

- __iter__

  返回 self ，以便在应该使用可迭代对象的地方使用迭代器，例如 for 循环中。

这个接口在 collections.abc.Iterable 抽象基类中制定。

因为迭代器只需要 __next__ 和 __iter__ 两个方法，所以除了调用 next() 方法，以及捕获 StopIteration 异常之外，没有办法检查是否还有遗留的元素。此外，也没有办法“还原”迭代器。如果想再次迭代，就要调用 iter(...) ，传入之前构建迭代器得可迭代对象。传入迭代器本身没有用，因为 Iterator.__iter__ 方法的实现方式是返回实例本身，所以传入迭代器无法还原已经耗尽的迭代器。

```
# abc.Iterator 类
class Iterator(Iterable): # Iterable 只有 __iter__ 一个内置函数
    __slots__ = ()

    @abstractmethod
    def __next__(self):
        raise StopIteration

    def __iter__(self):
        return self

    @classmethod
    def __subclasshook__(cls, C):
        if cls is Iterator:
            if (any("__next__" in B.__dict__ for B in C.__mro__) and
                any("__iter__" in B.__dict__ for B in C.__mro__)):
                return True
            return NotImplemented
```

现在我们总结一下可迭代对象与迭代器：

- 可迭代对象

使用 `iter` 内置函数可以获取迭代器的回想。如果对象实现了能返回迭代器的 `__iter__` 方法，那么对象是可迭代。序列都可以迭代；实现了 `__getitem__` 方法，而且其参数是从零开始的索引，这种对象也可以迭代。

可迭代对象一般通过 `iter(x)` 进行检验。

- 迭代器

实现了无参数的 `__next__` 方法，返回序列的下一个元素；如果没有元素了，那么抛出 `StopIteration` 异常。Python 中的迭代器还实现了 `__iter__` 方法，因此迭代器也可以迭代。

迭代器对象一般通过 `isinstance(x, abc.Iterator)` 进行检验。

14.3 Sentence 类第 2 版：典型的迭代器

这一版的对象实现典型的迭代器涉及模型，但并不符合 Python 习惯做法，后面重构会说明原因。这一版能明确可迭代的集合和迭代器对象之间的关系。

```
# 例14-4: Sentence 第2版
import re
import reprlib

RE_WORD = re.compile('\w+')

class Sentence:

    def __init__(self, text):
        self.words = RE_WORD.findall(text)
        self.text = text

    def __repr__(self):
        return 'Sentence(%s)' % reprlib.repr(self.text)

    def __iter__(self):   # 与上一版比，没有 __getitem__
        return SentenceIterator(self.words) # 实例化并返回一个迭代器

class SentenceIterator:

    def __init__(self, words):
        self.words = words
        self.index = 0

    def __next__(self):
        try:
            word = self.words[self.index]
        except IndexError:
            raise StopIteration
        self.index += 1
        return word
```

```
32
33      def __iter__(self):
34          return self
```

对于上面示例的 SentenceIterator 来说，没有必要实现 __iter__ 方法，因为几乎不会使用到该方法，实现的目的是为了满足迭代器协议，而且这么做能让迭代器通过 issubclass(SentenceIterator, abc.Iterator) 测试。如果让 SentenceIterator 类继承 abc.Iterator，那么它会继承对应的 __iter__ 方法。

把 Sentence 编程迭代器：坏主意

构建可迭代对象与迭代器时经常会出现错误，我们必须注意，迭代器可以迭代，但是可迭代对象不是迭代器。原因是可迭代对象的有个 __iter__ 方法，每次都实例化一个新的迭代器；而迭代器要实现 __next__ 方法，返回单个元素，此外还要实现 __iter__ 方法，返回迭代器本身。

如果我们在 Sentence 类中实现 __next__ 方法，让 Sentence 实例既是可迭代对象，也是自身的迭代器。可是这种方法是十分糟糕的，绝对不要这样使用。其主要原因是支持多种便利，即必须从同一个迭代实例中获取多个独立的迭代器，而且各个迭代器要维护自身的内部状态。

14.4 Sentence 类第 3 版：生成器函数

实现相同功能，但却符合 Python 习惯的方式是，用生成器函数代替 SentenceIterator 类。

```
1  # 例14-5: 第3版Sentence类
2  import re
3  import reprlib
4
5  RE_WORD = re.compile('\w+')
6
7  class Sentence:
8
9      def __init__(self, text):
10         self.text = text
11         self.words = RE_WORD.findall(text)
12
13     def __repr__(self):
14         return 'Sentence(%s)' % reprlib.repr(self.text)
15
16     def __iter__(self):
17         for word in self.words:
18             yield word     # 产出当前 word
19         return     # 可以不写
```

在上述示例中，迭代器其实是生成器对象，每次调用 __iter__ 方法都会自动创建，因为这里的 __iter__ 方法是生成器函数。

生成器函数的工作原理

只要 Python 函数的定义体中有 `yield` 关键字，该函数就是生成器函数。调用生成器函数时，会返回一个生成器对象。也就是说，生成器函数是生成器工厂。

生成器函数会创建一个生成器对象，包装生成器函数的定义体。把生成器传给 `next(...)` 函数时，生成器函数会向前，执行函数定义体中的下一个 `yield` 语句，返回产出的值，并在函数定义体的当前位置暂停。最终，函数的定义体返回时，外层的生成器对象会抛出 `StopIteration` 异常——这一点与迭代器协议一致。

注意，生成器对象不需要 `return` 语句，如果有并遇到了 `return` 则会抛出 `StopIteration` 错误。

下面函数将更佳清楚地说明生成器函数工作原理。

```
>>> def gen_AB():     # 定义生成器函数与普通函数无异
...     print('start')
...     yield 'A'          # 第一次调用 __next__ 时，停在此处
...     print('continue')
...     yield 'B'          # 第二次调用停在此处
...     print('end.')      # 第三次调用，打印 end. 并抛出 StopIteration 异常
...
>>> for c in gen_AB():
...     print('-->', c)
...
start
---> A
continue
---> B
end.
>>>         # for 循环捕捉了 StopIteration 异常，因此没有报错
```

14.5 Sentence 类第 4 版：惰性实现

惰性在编程语言和 API 中被认为是好的特性。惰性实现是指尽可能延后生成值。这样做能节省内存，而且获取还可以避免做无用的处理。

设计 `Iterator` 接口时考虑到了惰性：`next(my_iterator)` 一次生成一个元素。目前实现的几版 `Sentence` 类都不具有惰性，因为 `__init__` 方法急迫地构建好了文本中的单词列表，然后将其绑定到 `self.words` 属性上。这样就得处理整个文本，降低了效率。

`re.finditer` 函数是 `re.findall` 函数的惰性版本，返回的不是列表，而是一个生成器，按需生成 `re.MatchObject` 实例，这将节省大量内存。

```
# 例14-7: 第4版Sentence类
import re
import reprlib

RE_WORD = re.compile('\w+')

```

```
class Sentence:

    def __init__(self, text):
        self.text = text      # 这里不构建 words

    def __repr__(self):
        return 'Sentence(%s)' % reprlib.repr(self.text)

    def __iter__(self):
        # re.finditer 函数构建一个迭代器，包含 self.text 中匹配 RE_WORD 的单词，返回
            MatchObject 实例
        for match in RE_WORD.finditer(self.text):
            yield match.group() # 从 MatchObject 实例中提取匹配正则表达式的具体文本
```

14.6 Sentence 类第 5 版：生成器表达式

简单的生成器函数，可以替换成生成器表达式。

生成器表达式可以理解为列表推导的惰性版本：不会迫切地构建列表，而是返回一个生成器，按需惰性生成元素。也就是说，如果列表推导是制造列表的工程，那么生成器表达式就是制造生成器的工厂。

生成器表达式会产出生成器，由此可以构建更为精简的 Sentence 类：

```
# 例14-7: 第5版Sentence类
import re
import reprlib

RE_WORD = re.compile('\w+')

class Sentence:

    def __init__(self, text):
        self.text = text

    def __repr__(self):
        return 'Sentence(%s)' % reprlib.repr(self.text)

    def __iter__(self):
        return (match.group() for match in RE_WORD.finditer(self.text))
```

14.7 何时使用生成器表达式

生成器表达式是语法糖：完全可以替换成生成器函数，不过有时生成器表达式更佳便利。生成器表达式是创建生成器的简洁句法，这样无需先定义函数再调用。不过，生成器灵

活得多，可以使用多个语句实现复杂的逻辑，也可以作为协程[1]使用。

选择何种句法很容易判断：如果生成器表达式要分成多行写，倾向于定义生成器函数，以便提高可读性。此外生成器函数可以重用。

如果函数或构造方法只有一个参数，传入生成器表达式时不用写一对调用函数的括号，再写一对生成器表达式的阔含，只写一对就行了：

```
    return Vector(n * scalar for n in self)
```

14.8 等差数列生成器

下面我们来创建一个等差数列生成类：

```
# 例14-11: 等差数列生成器
class ArithmeticProgression:

    def __init__(self, begin, stop, end=None):
        self.begin = begin
        self.step = step
        self.end = end # None -> 无穷数列

    def __iter__(self):
        # 这一段将 self.begin 传给 result，不过会先强制转换成前面的加法算式得到的类型
        result = type(self.begin + self.step)(self.begin)
        forever = self.end is None
        index = 0
        while forever or result < self.end:
            yield result
            index += 1
            result = self.begin + self.step * index #
                由于算术运算符会隐式转换数值类型，这里没有必要强制转换
```

在最后一行，我们没有使用 self.step 不断地增加 result 而是应用了 index 获得，这样避免了处理浮点数时累积效应引起的风险。

在这个类中，我们应用生成器表达式的特性，实现了无穷数列的生成。

这个示例简单地说明了如何使用生成器函数实现特殊的 `__iter__` 方法。不过如果一个类只是为了构建生成器而去实现 `__iter__` 方法，那还不如使用生成器函数。

```
# 例14-12: 效果相同的生成器函数
def aritprog_gen(begin, step, end=None):
    result = type(begin+step)(begin)
    forever = end is None
    index = 0
    while forever or result < end:
        yield result
        index += 1
```

[1]后文会讲。

```
        result = begin + index * step
```

使用 itertools 模块生成等差数列

Python3.4 中的 itertools 模块提供了 19 个生成器函数，结合起来使用能实现很多有趣的用法。

例如，itertools.count 函数返回的生成器能生成多个数，如果不传入参数，生成从零开始的整数数列。也可以提供可选的 start 和 step 值。

```
>>> import itertools
>>> gen = itertools.count(1, .5)
>>> next(gen)
1
>>> next(gen)
1.5
```

然而，像 itertools.count 这样的生成无穷个元素的函数从不停止。因此，如果调用 list(count())，Python 会创建一个特别大的表，导致电脑崩溃。

不过，itertools.takewhile 函数则不同，它会生成一个使用另一个生成器的生成器，在指定的条件计算结果为 False 时停止。因此，可以把这两个函数结合起来使用，编写下述代码：

```
>>> gen = itertools.takewhile(lambda n: n<3 , itertools.count(1, .5))
>>> list(gen)
[1, 1.5, 2.0, 2.5]
```

下面使用这两个函数，写出有穷的等差数列函数：

```
# 例14-13: 有穷等差数列
import itertools


def aritprog_gen(begin, step, end):
    first = type(begin+step)(begin)
    ap_gen = itertools.count(first, step)
    if end is not None:
        ap_gen = itertools.takewhile(lambda n: n < end, ap_gen)
    return ap_gen
```

注意上述函数不是生成器函数，而是生成器工厂函数。

14.9 标准库中的生成器函数

标准库中提供了很多生成器，下面将按功能分组，简单介绍其中的 24 个。[2]

第一组是用于过滤的生成器函数：从输入的可迭代对象中产出元素的子集，而且不修改元素本身。就像 itertools.takewhile 函数一样，下表中的大多数函数都接受一个断言参数。

[2]这里每个函数只提供了一个示例，更全面的用法请看原书 P350 或参考官方文档。

这个参数是个布尔函数，有一个参数，会应用到输入中的每个元素上，用于判断元素是否包含在输出中。

表 5.1 用于过滤的生成器函数

模块	函数	说明
itertools	compress(it, selector_it)	并行处理两个可迭代的对象，如果 selector_it 中的元素是真值，产出 it 中对应的元素。
itertools	dropwhile(predicate, it)	处理 it 跳过 predicate 的计算结果为真值的元素，然后产出剩下的各个元素。(不再进一步检查)
(内置)	filter(predicate, it)	把 it 中的各个元素传给 predicate，如果 predict(item) 返回真值，那么产出对应的元素；如果 predict 是 None，那么只产生真值元素。
itertools	filterfalse(predicate, it)	与 filter 函数的作用类似，不过 predicate 的逻辑是相反的：predicate 返回假值时产出对应的元素
itertools	islice(it,start,stop,step)	产出 it 的切片，作用类似于 s[:stop] 或 s[start:stop:step]，不过 it 可以是任何可迭代的对象，而且这个函数实现的是惰性操作。
itertools	takewhile(predicate, it)	predicate 返回为真值时产出对应元素，然后立即停止，不再继续检查。

下面演示各个函数的用法：

```
>>> import itertools
>>> def vowel(c):
...     return c.lower() in 'aeiou'
...
>>> list(filter(vowel, 'Aardvark')) # 返回真值则产生
['A','a','a']
>>> list(itertools.filterfalse(vowel,'Aardvark')) # 返回假值则产生
['r','d','v','r','k']
>>> list(itertools.dropwhile(vowel,'Aardvark')) # 跳过前面的真值
['r','d','v','a','r','k']
>>> list(itertools.takewhile(vowel,'Aardvark')) # 只产生前面的真值
['A','a']
>>> list(itertools.compress('Aardvark',(1,0,1,1,0,1))) # 产生对应为真的值
['A','r','d','a']
>>> list(itertools.islice('Aardvark',1,7,2)) # 类似切片操作
['a','d','a']
```

下一组用于映射的生成器函数：在输入的单个可迭代对象中的各个元素上做计算，然后返回结果。

表 5.2 用于映射的生成器函数

模块	函数	说明
itertools	accumulate(it,[func])	产出累计的总和：如果提供了 func, 那么把前两个元素传给它，然后把结果和下一个元素运算，以此类推。
(内置)	enumerate(iterable, start=0)	产出由两个元素组成的元组，结构是 (index,item) , 其中 index 从 start 开始计数，item 则从 iterable 中取。
(内置)	map(func,it1,[it2...])	把 it 中的各个元素传给 func 产出结果；如果传入 N 个可迭代对象，那么 func 必须能接受 N 个参数，而且要并行处理各个可迭代对象。
itertools	starmap(func,it)	把 it 中的各个元素传给 func 产出结果；输入的可迭代对象应该产出可迭代元素 iit，然后以 func(*itt) 这种形式调用 func。

```
>>> import itertools
>>> import operator
>>> sample = [5,4,2,8,7,6,3,0,9,1]
>>> list(itertools.accumulate(sample,max)) # 产生最值
[5,5,5,8,8,8,8,8,9,9]
>>> list(enumerate('abc',1)) # 产生编号与元素
[(1,'a'),(2,'b'),(3,'c')]
>>> list(map(operator.mul, rang(11), range(11))) # 映射运算
[0,1,4,9,16,25,36,49,64,81,100]
>>> list(itertools.starmap(operator.mul,enumerate('abc',1))) # 拆包映射运算
['a','bb','ccc']
```

接下来一组用于合并的生成器函数，这些函数都从输入的多个可迭代对象中产出元素。

表 5.3 合并多个了迭代对象的生成器函数

模块	函数	说明
itertools	chain(it1,...)	先产出 it1 中的所有元素，然后产出 it2 中的所有元素，以此类推。
itertools	chain.from_iterable(it)	产出 it 生成的各个可迭代对象的元素，一个接一个，无缝衔接在一起。
itertools	product(it1,...,repeat=1)	计算笛卡尔积；repeat 指明重复处理多少次输入的可迭代对象。
(内置)	zip(it1,...)	并行从输入的各个可迭代对象中获取元素，产出 N 个元素组成的元组。
itertools	zip_longest(it1,..., fillvalue=None)	与上一个函数类似，但持续到最长的对象，空缺值使用 fillvalue 填充。

```
>>> import itertools
```

```
>>> list(itertools.chain('ABC', range(2))) # 依次产出元素
['A','B','C',0,1]
>>> list(itertools.chain.from_iterable(enumerate('ABC'))) # 从生成其中依次产出元素
[0,'A',1,'B',2,'C']
>>> list(zip('ABC',range(5))) # 并行取出元素
[('A',0),('B',1),('C',2)]
>>> list(itertools.zip_longest('ABC',range(5))) # 并行取出(最长)
[('A',0),('B',1),('C',2),(None,3),(None,4)]
>>> list(itertools.product('ABC',range(2))) # 笛卡尔积
[('A',0),('A',1),('B',0),('B',1),('C',0),('C',1)]
```

有些生成器函数会从一个元素中产出多个值，扩展输入的可迭代对象：

表 5.4 把输入的各个元素扩展成多个输出元素的生成器函数

模块	函数	说明
itertools	combinations (it,out_len)	把 it 产出的 out_len 个元素组合在一起，然后产出
itertools	combinations_with_replacement(it,out_len)	把 it 产出的 out_len 个元素组合在一起，然后产出，包含相同元素的组合
itertools	count(start=0,step=1)	从 start 开始不断产出数字，按 step 指定的步幅增加
itertools	cycle(it)	从 it 中产出各个元素，存储各个元素的副本，然后按顺序重复不断地产出各个元素
itertools	permutations it,out_len=None	把 out_len 个 it 产出的元素排列在一起，然后产出这些排列；out_len 的默认值等于 len(list(it))
itertools	repeat(item,[times])	重复不断地产出指定的元素，除非提供 items，指定次数。

演示 count, repeat, cycle 的用法：

```
>>> ct = itertools.count()   # 累加
>>> next(ct), next(ct)
(0,1)
>>> cy = itertools.cycle('ABC')  # 循环
>>> list(itertools.islice(cy,7))
['A','B','C','A','B','C','A']
>>> rp = itertools.repeat(7) # 重复
>>> next(rp), next(rp)
(7,7)
```

组合生成器 (其余三个生成器函数) 用法：

```
>>> list(itertools.combinations('ABC',2))
[('A','B'),('A','C'),('B','C')]
>>> list(itertools.combinations_with_replacements('ABC',2))
[('A','A'),('A','B'),('A','C'),('B','B'),('B','C'),('C','C')]
>>> list(itertools.product('ABC', repeat=2))
[('A','A'),('A','B'),('A','C'),('B','A'),('B','B'),('B','C'),('C','A'),('C','B'),('C','C')]
```

最后一组生成器函数用于产出输入的可迭代对象中的全部元素，不过会以某种方式重新排列。

表 5.5 用于重新排列元素的生成器函数

模块	函数	说明
itertools	groupby(it,key=None)	产出由两个元素组成的元素，形式为 (key,group)，其中 key 是分组标准，group 是生成器，用于产出分组里的元素。
(内置)	reversed(seq)	从后向前，倒序产出 seq 中的元素，seq 必须是序列，或者实现了 __reversed__ 特殊方法的对象。
itertools	tee(it,n=2)	产出一个由 n 个生成器组成的元组，每个生成器用于单独产出输入的可迭代对象中的元素。

itertools.groupby 生成器函数的用法

```
>>> list(itertools.groupby('LLLLAAGGG'))
[('L', <itertools._grouper object at 0x000001CF2156C7C8>),
('A', <itertools._grouper object at 0x000001CF2156C848>),
('G', <itertools._grouper object at 0x000001CF2156C8C8>)]
>>> animals = ['duck','eagle','rat','giraffe','bear','bat','dolphin','shark','lion']
>>> animals.sort(key=len)
>>> for length,group in itertools.groupby(animals,len):
...     print(length, '->', list(group))
3 -> ['rat', 'bat']
4 -> ['duck', 'bear', 'lion']
5 -> ['eagle', 'shark']
7 -> ['giraffe', 'dolphin']
```

itertools.tee 生成器函数的用法

```
>>> list(itertools.tee('ABC'))
[<itertools._tee object at 0x000001CF21572108>, <itertools._tee object at 0x000001CF21572148>]
>>> g1, g2 = itertools.tee('ABC')
>>> next(g1)
'A'
>>> next(g2)
'A'
```

14.10 新句法 yield from

yield from 是 Python3.3 中新出现的句法。如果生成器函数需要产出一个生成器生成值，传统的解决方法是使用嵌套的 for 循环。[3]

```
>>> def chain(*iterables):
...     for it in iterables:
```

[3]标准库中的 chain 函数是用 C 语言写的。

```
...         for i in it:
...             yield i
...
>>> s = 'ABC'
>>> t = tuple(range(3))
>>> list(chain(s,t))
['A','B','C',0,1,2]
```

chain 生成器函数把操作依次交给接收到的各个可迭代对象处理。为此，引入了一个新句法，如下述代码所示：

```
>>> def chain(*iterables):
...     for it in iterables:
...         yield from it
```

可以看出，yield from 完全代替了内层的 for 循环。除了代替循环之外，yield from 还会创建通道，把内层生成器直接与外层生成器的客户端联系起来。[4]

14.11 可迭代的归约函数

归约函数接受一个可迭代的对象，然后返回单个结果。下表列出的规约函数其实都可以使用 functools.reduce 函数实现，内置是因为使用它们便于解决常见的问题。此外，对 all 和 any 函数来说，有一项重要的优化措施是 reduce 函数做不到的：这两个函数会短路。(一旦确定解决立马停止计算)

表 5.6 读取迭代器，返回单个值的内置函数

模块	函数	说明
(内置)	all(it)	it 中所有元素都为真时返回 True，空值返回 True
(内置)	any(it)	it 中存在元素为真时返回 True，空值返回 False
(内置)	max(it,[key=,] [default=])	返回最大元素，key 是排序函数，如果可迭代对象为空，返回 default
(内置)	min(it,[key=,] [default=])	返回最小元素，key 是排序函数，如果可迭代对象为空，返回 default
functools	reduce(func,it, [initial])	把前两个元素传给 func，然后把计算结果和第三个元素传给 func；如果提供了 initial 则将其作为第一个元素。
(内置)	sum(it,start=0)	计算总和，如果提供了 start 会把它加上。

还有一个内置的函数接受一个可迭代对象，返回不同的值——sorted。reversed 是生成器函数，与此不同，sorted 会构建并返回真正的列表。

[4]后文会有更详细的说明。

14.12 深入分析 iter 函数

如前所述，在 Python 中迭代对象 x 时会调用 `iter(x)`。

`iter` 函数还有一个鲜为人知的用法：传入两个参数，使用常规函数或任何可调用的对象创建迭代器。这样使用时，第一个参数必须是可调用对象，用于不断调用，产出值；第二个参数是哨符，这是个标记值，当可调用的对象返回这个值时，触发迭代器抛出 `StopIteration` 异常，而不产出哨符。

应用这个特性，我们构建一个致骰子游戏，知道掷出 1 点为止：

```
>>> def d6():
...     return randint(1,6)
...
>>> d6_iter = iter(d6,1)
>>> for roll in d6_iter:
...     print(roll)
...
4 3 6 3
```

这里绝对不对产生 1，因为 1 是我们的哨符。

内置函数 `iter` 的文档中有个使用的例子。这段代码逐行读取文件，直到遇到空行或者到达文件末尾位置：

```
with open('mydata.text') as fp:
    for line in iter(fp.readline, '\n'):
        process_line(line)
```

本章到这里结束，原书 P360 页还有对数据处理方面的介绍。

15 上下文管理器和 else 块

15.1 if 语句之外的 else 块

else 子句不仅能在 if 语句中使用，还能在 for,while,try 语句中使用：

- for

仅当 for 循环运行完毕时 (for 循环没有被 break 语句中止) 才运行 else 块。

- while

仅当 while 循环因为条件为假值而退出时 (即 while 循环没有被 break 语句中止) 才运行 else 块。

- try

仅当 try 块中没有异常抛出时才运行 else 块。且 else 子句抛出的异常不会由前面的 except 子句处理。

在所有情况下，如果异常或者 return,break,continue 语句导致控制权跳到了复合语句的主块之外，else 子句也会跳过。

下面会讲解 else 语句的具体案例，但原作者并不推荐在 if 语句之外的地方使用 else 关键字，因为 else 的字面意思具有 “排他性” 这层意思。但是在 if 语句之外的语句中 else 往往代表着然后，其意义更接近 then ，这会造成很多误解。

在这些语句中使用 else 子句通常能让代码易于理解，而且能省去一些麻烦，不用设置控制标志或者条件额外的 if 语句。

在循环中使用 else 子句的方式如下述代码片段所示：

```
for item in my_list:
    if item.flavor == 'banana':
        break
else:
    raise ValueError('No banana flavor found!')
```

在 try/except 语句中使用 else 如下述代码片段所示：

```
try:
    dangerous_call()
except OSError:
    log('OSError...')
else:
    after_call()
```

上述代码中 try 防守的是 dangerous_call() 可能出现的错误，而不是 after_call()，只有 try 块不抛出异常，才会执行 after_call()。

在 Python 中，try/except 不仅用于处理错误，还常用于控制流程。为了 Python 官方词汇表还定义了一个缩略词。

- EAFP (easier to ask for forgiveness than permission)

获得原谅比获得许可容易。这是一种常见的 Python 编程风格，先假定存在有效的键或属性，如果假定不成立，那么捕获异常。这种风格简单明快，特点是代码中有很多 `try/except` 语句。与其它语言一样，这种风格的对立面是 LBYL 风格。

- LBYL (look before you leap)

  三思而后行。这种编程风格在调用函数或查找属性或健之前显式测试前提条件。特点是代码中有很多 `if` 语句。

15.2 上下文管理器和 `with` 块

上下文管理器对象存在的目的是管理 `with` 语句，就像迭代器的存在是为了管理 `for` 语句一样。

`with` 语句的目的是简化 `try/finally` 模式。这种模式用于保证一段代码运行完毕后执行某项操作，即便那段代码由于异常，`return` 语句或 `sys.exit()` 调用而中止，也会执行指定的操作。`finally` 子句中的代码通常用于释放重要的资源，或者还原临时变量的状态。

上下文管理器协议包含 `__enter__` 和 `__exit__` 两个方法。with 语句开始运行时，会在上下文管理器对象上调用 `__enter__` 方法。运行结束后调用 `__exit__` 方法，依次扮演 `finally` 子句的角色。

最常见的例子是确保关闭文件对象：

```
>>> with open('mirror.py') as fp: # fp 绑定到打开的文件上，因为文件的 __enter__ 方法返回 self
...     src = fp.read(60) # 从 fp 中读取一些数据
...
>>> len(src)   # fp 对象仍可用
60
>>> fp.closed, fp.encoding # 读取 fp 对象的属性
(True, 'UTF-8')
>>> fp.read(60)   # 不能进行 I/O 操作，因为 with 块的末尾，文件被关闭了
Traceback ...
```

上面示例的第一行道出了不易察觉但很重要的一点：执行 `with` 后面的表达式得到的结果是上下文管理器对象，不过，把值绑定到目标变量商是在上下文管理器对象商调用 `__enter__` 方法的结果。

碰巧，上述示例中的 open() 函数返回 TextUIWrapper 类的实例，而该实例的 `__enter__` 方法返回 self。不过，`__enter__` 方法除了会返回上下文管理器之外，还可能返回其他对象。

不管控制流程以哪种方式退出 `with` 块，都会在上下文管理器对象上调用 `__exit__` 方法，而不是在 `__enter__` 方法返回的对象上调用。

`with` 语句的 `as` 子句是可选的，对 `open` 函数来说，必须加上 `as` 子句，以便获取文件的引用。不过，有些上下文管理器会返回 `None`，因为没什么有用的对象能提供给用户。

下面是一个精心制作的上下文管理器执行操作，以此强调上下文管理器与 `__enter__` 方法返回的对象之间的区别。

```
>>> from mirror import LookingGlass
>>> with LookingGlass() as what: # 返回结果绑定到 what 上
...     print('Alice, Kitty and Snowdrop')
...     print(what)
...
pordwonS dna yttiK ,ecilA
YKCOWREBBAJ
>>> what   # with 执行结束，可以获得 __enter__ 方法的返回值，即存储在 what 变量中的值
'JABBERWOCKY'
>>> print('Back to normal.') # 输出不再是反向
Back to normal.
```

下面是 LookingGlass 类的实现：

```
# 例15-3: LookingGlass 类
class LookingGlass:

    def __enter__(self):  # enter 方法只传入 self
        import sys
        self.original_write = sys.stdout.write # 将原来的输出保存
        sys.stdout.write = self.reverse_write # 打猴子补丁，替换成自编写的方法
        return 'JABBERWOCKY'  # 返回值

    def reverse_write(self, text):
        self.original_write(text[::-1])

    # 如果一切正常，传入的参数是 None, None, None; 否则返回对应异常
    def __exit__(self, exc_type, exc_value, traceback):
        import sys     # 重复导入模块不会消耗很多的资源，因为 Python 会缓存导入的模块
        sys.stdout.write = self.original_write # 还原成原来的 sys.stdout.write
        if exc_type is ZeroDivisionError: # 若有异常，打印一个消息
            print('Please DO NOT divide by zero!')
            return True # 返回 True 告诉编译器已经处理了，否则，向上冒泡告诉编译器
```

解释器调用 __enter__ 方法时，除了隐式的 self 之外，不会传入任何参数。传给 __exit__ 方法的三个参数列举如下：

- exc_type

  异常类 (例如 ZeroDivisionError)

- exc_value

  异常实例。有时会有参数传给异常构造方法，例如错误消息，这些参数可以使用 exc_value.args 参数

- traceback

  traceback 对象。

上下文管理器是相当新颖的特性，更多请查阅官方文档。

15.2.1 contextlib 模块中的实用工具

自己定义上下文管理器之前，先看一下 Python 标准文档中的 contextlib 内容。contextlib 模块中有一些类的函数，使用范围十分广泛。

- closing

  如果对象提供了 close() 方法，但没有实现 __exnter__/__exit__ 协议，那么可以使用这个函数构建上下文管理器。

- suppress

  构建临时忽略指定异常的上下文管理器。

- @contextmanager

  这个装饰器把简单的生成器函数变成上下文管理器。这种上下文管理器也能用于装饰函数，在受管理的上下文中运行整个函数。

- ContextDecorator

  这是个基类，用于定义基于类的上下文管理器。这种上下文管理器也能用于装饰函数，在受管理的上下文中运行整个函数。

- ExitStack

  这个上下文管理器能进入多个上下文管理器。with 块结束时，ExitStack 按照后进先出的顺序调用栈中各个上下文管理器的 __exit__ 方法。如果事先不知道 with 块要进入多少个上下文管理器，可以使用这个给类。例如，同时打开任意一个文件列表中的所有文件。

在这些实用工具中，使用最广泛的是 @contextmanager 装饰器。

15.3 使用 @contextmanager

@contextmanager 装饰器能减少创建上下文管理器的样板代码量，因为不用编写一个完整的类，定义 __enter__ 和 __exit__ 方法，而只需实现有一个 yield 语句的生成器，生成想让 __enter__ 方法返回的值。

在使用 @contextmanager 装饰的生成器中，yield 语句的作用是把函数的定义分成两部分：yield 语句前面的所有代码在 with 块开始时 (即解释器调用 __enter__ 方法时) 执行，后面的代码在 with 块结束时 (即解释器调用 __exit__ 方法时) 执行。

下面这个例子重新编写了 LookingGlass 类：

```
# 例15-5: 使用 @contextmanager 编写的 LookingGlass 类
import contextlib

@contextlib.contextmanager
def looking_glass():
    import sys
    original_write = sys.stdout.write

    def reverse_write(text):
```

```
        original_write(text[::-1])

    sys.stdout.write = reverse_write
    yield 'JABBERWOCKY' # 产出一个值，这个值会绑定到 with 语句中的 as 子句的目标变量上。执行
        with 模块时，函数将在这一点暂停
    sys.stdout.write = original_write
```

这个生成器的用法和类没有什么区别。本质上，`contextlib.contextmanager` 装饰器会把函数包装是实现 `__enter__` 和 `__exit__` 方法的类。

这个类的 `__enter__` 方法有如下作用：

1. 调用生成器，保存生成器对象 (这里称为 `gen`)。
2. 调用 `next(gen)`，执行到 `yield` 关键字所在的位置。
3. 返回 `next(gen)` 产出的值，以便把产出的值绑定到 `with/as` 语句中的目标变量上。

`with` 块终止时，`__exit__` 方法会做以下几件事。

1. 检查有没有把异常传给 `exc_type`; 如果有，调用 `gen.throw(exception)`, 在生成器函数定义体中包含 `yield` 关键字的那一行抛出异常。
2. 否则，调用 `next(gen)`, 继续执行生成器函数定义体中 `yield` 语句之后的代码。

这样看来，示例 15-5 出了一个严重的错误：如果在 `with` 块中抛出异常，Python 解释器会将其捕获，然后再 `looking_glass` 函数的 `yield` 表达式中再次抛出。但是，那里没有处理错误的代码，因此 `looking_glass` 函数会终止，永远无法恢复成原来的 `sys.stdout.write` 方法，导致系统处于无效状态。

下面添加一些代码，这样在功能上它就与示例 15-3 中基于类的实例等效了：

```
# 示例15-5: 改进后的使用 @contextmanager 编写的 LookingGlass 类
import contextlib

@contextlib.contextmanager
def looking_glass():
    import sys
    original_write = sys.stdout.write

    def reverse_write(text):
        original_write(text[::-1])

    sys.stdout.write = reverse_write
    msg = ''   # 用于保存错误信息
    try:
        yield 'JABBERWOCKY'
    except ZeroDivisionError:
        msg = 'Please DO NOT divide by zero!'
    finally:
        sys.stdout.write = original_write
        if msg:
            print(msg)
```

前面说过，为了告诉解释器异常已经处理了，`__exit__` 方法会返回 `True`，此时解释器会压

制异常。如果 `__exit__` 方法没有显式返回一个值，那么解释器得到的是 `None`，然后向上冒泡异常。使用 `@contextmanager` 装饰器时，默认的行为是相反的：装饰器提供的 `__exit__` 方法假定发给生成器的所有异常都得到了处理，因此应该压制异常[5]。如果不想让 `@contextmanager` 压制异常，必须在被装饰的函数中显式重新抛出异常。[6]

使用 `@contextmanager` 装饰器时，要把 `yield` 语句放在 `try/finally` 语句中 (或者 `with` 语句中)，这是无法避免的，因此我们永远不知道上下文管理器中在做什么。

[5]把异常发给生成器的方式是使用 `throw` 方法，下一章会说明。

[6]这样约定的原因是，创建上下文管理器时，生成器无法返回值，只能产出值。下一章会有更详细讲解。